(2)

RESEARCH TRIANGLE INSTITUTE

ARTIFICIAL INTELLIGENCE FOR VHSIC SYSTEMS DESIGN (AIVD) USER REFERENCE MANUAL

December 1988

Prepared for:

Department of the Army
U.S. Army Electronics Research and Development Command
Fort Monmouth, New Jersey 07703-5000
Contract No. DAAL01-86-C-0040

Prepared by:

Center for Digital Systems Research
Research Triangle Institute
Research Triangle Park, North Carolina 27709

Octy, Inc.
10920 Oxford Court
Fairfax Station, Virginia 22039

105pl

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	DEC 1988	Software Users Manual

4. TITLE AND SUBTITLE
Artificial Intelligence for VHSIC Systems Design (AIVD) Users Reference Manual

5. FUNDING NUMBERS

6. AUTHOR(S)
Research Triangle Institute
OCTY, Inc.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
ETDL, LABCOM
Circuits & Subsystems Design Branch
Fort Monmouth, NJ 07703-5601

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
DAAL01-86-C-0040

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Same as Final Report

14. SUBJECT TERMS
VHSIC, Software/hardware codesign, Artificial Intelligence graph transformation, ADAS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
U	U	U	

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

ARTIFICIAL INTELLIGENCE FOR VHSIC SYSTEMS DESIGN (AIVD) USER REFERENCE MANUAL

December 1988

Department of the Army
U.S. Army Electronics Research and Development Command
Fort Monmouth, New Jersey 07703-5000
Contract No. DAAL01-86-C-0040

Prepared by:

Center for Digital Systems Research
Research Triangle Institute
Research Triangle Park, North Carolina 27709

Octy, Inc.
10920 Oxford Court
Fairfax Station, Virginia 22039

Contents

List of Figures

List of Tables

1. INTRODUCTION

1.1 Summary of the AIVD System

The Artificial Intelligence for VHSIC Systems Design (AIVD) project was undertaken in order to enhance the capabilities of the Architecture Design and Assessment System (ADAS)[1] and to provide additional support for the ADAS software/hardware codesign methodology. AIVD supports software/hardware codesign in several ways:

- AIVD assists the user in building complex software system models from libraries of generic algorithms.
- AIVD assists the user in transforming generic algorithm descriptions to meet hardware resource constraints and interacting software subsystem resource requirements.
- AIVD assists the user in defining software performance attributes in terms of mission parameters and architecture parameters.
- AIVD assists the user in building experiments to explore trade-offs across large design spaces.

AIVD uses artificial intelligence techniques to implement these capabilities:

- Rule-based programs for software to hardware allocation and for graph transformations.
- Transformation rules (in the form of context-sensitive graph grammar productions) for modifying the structure and attributes of graphs.
- Attribute grammars to define performance measures in terms of mission and hardware parameters and to back-annotate models with performance results.

[1] ADAS is a registered trademark of the Research Triangle Institute.

1.2 Conventions Used in this Manual

The following font conventions are used throughout this manual:

italics used to designate ADAS attributes, variables and terms of special interest.

`typewriter` used to designate system prompts and responses sent to the computer system's standard output, example file, node, and node template names, algorithms, and example ADL files.

bold used to designate input to invoke a software tool or to respond to a system's or a tool's prompt or query.

1.3 Related Documents

AIVD is based on the tools and methodology developed by Research Triangle Institute (RTI) for ADAS. The following manuals for ADAS are important reference books for AIVD users:

- The ADAS User Manual, Version 2.5
- The ADAS Training Manual, Basic Course
- The ADAS Training Manual, Advanced Course

There are several excellent references on system design. The reference that most closely fits with the AIVD design philosophy is Bowen and Brown, "Systems Design," Volume II of *VLSI Systems Design for Digital Signal Processing.* Another important reference on searching large design spaces is T. Carpenter and S. Yalamanchili, "Rapid Evaluation of Parallel Architectures: An ADAS Implementation," *Presented at GOMAC '87*, (October, 1987).

2. AIVD OVERVIEW

2.1 The AIVD System

The AIVD system structure is shown in Figure 2.1. The system consists of four major tools interacting with four major data bases and the existing commercial ADAS tool set. The tools and data bases are described in the following sections.

2.2 The AIVD Tools

2.2.1 The Intelligent User Interface

The Intelligent User Interface (IUI) guides the user to select appropriate algorithms from the Application Domain Hierarchy. The IUI provides a browsing capability for exploring hierarchical libraries of generic algorithms. This includes viewing the graph and template hierarchies associated with a library and reading the help files associated with the library. It also includes an object-oriented editing capability for hierarchical models, so that the user can point to an object and edit text files associated with the object, such as an ADL program, Ada or C language source code files, and help files.

2.2.2 The Graph Transformation System

The Graph Transformation System (GTS) aids the user in customizing software to fit system constraints, including the capabilities of available hardware resources and the processing requirements of other algorithms that are part of the system model.

Transformations define how to change an algorithm to improve its fit with the system constraints without changing its function. Transformations can be used to increase or decrease parallelism, to insert fault-tolerant features into an algorithm, to represent the cost of communications delays, or to eliminate unnecessarily redundant operations from an algorithm's description.

2.2.3 The Attribute Definition Language Evaluator

The Attribute Definition Language Evaluator (ADLEVAL) translates ADL programs into script files which can be read by the other AIVD tools, including the ADAS editor and simulator. The script files set the performance attributes of the ADAS models, including node *firing_delay* and *module_class*, inport *token_consume_rate* and *firing_threshold*, and outport *token_produce_rate*.

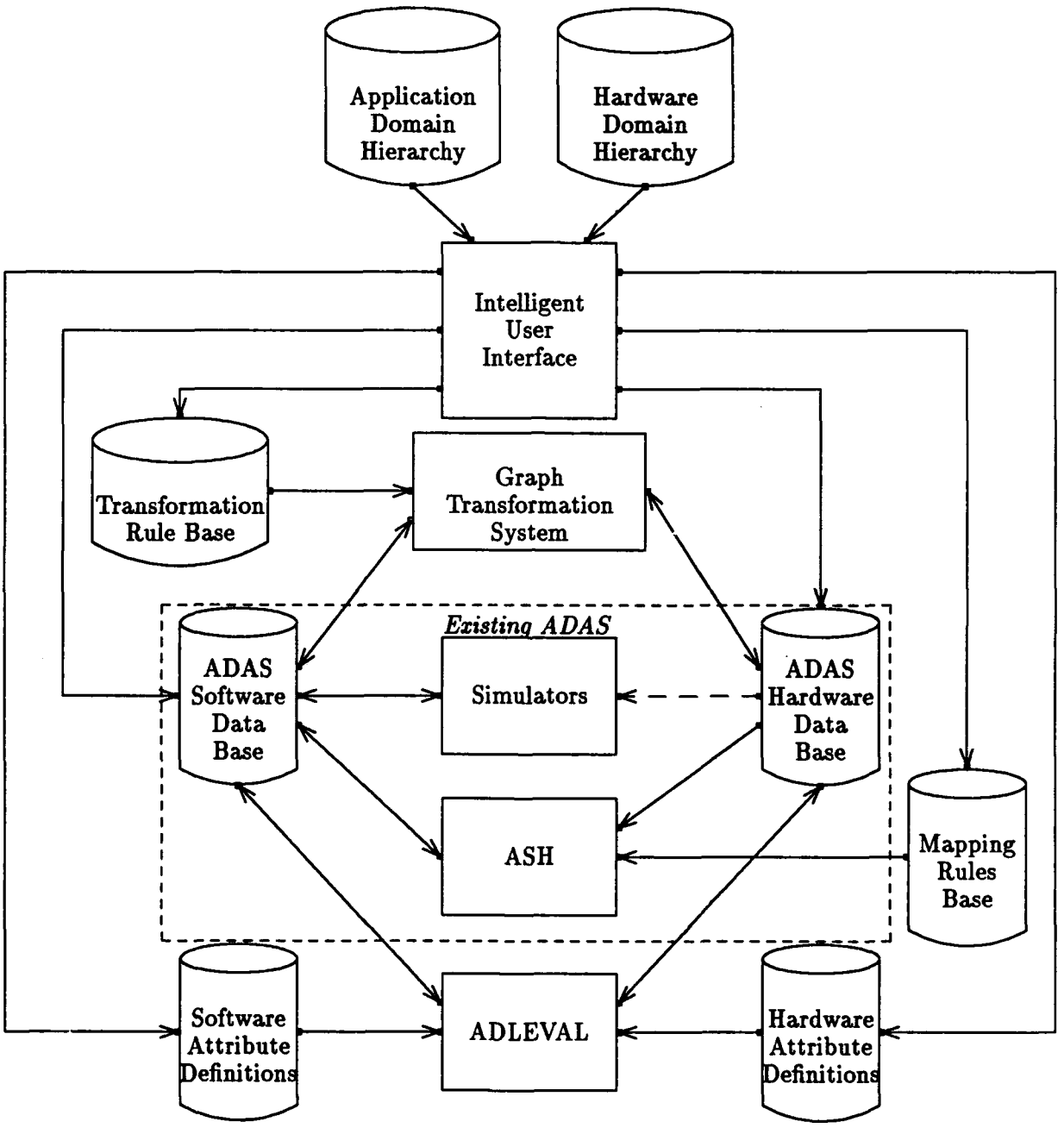

Figure 2.1: AIVD System Structure

2.2.4 The ADAS Simulators

The AIVD tool set is compatible with the three ADAS simulators: GIPSIM, CSIM, and ADASIM. GIPSIM is a directed graph simulator, while CSIM and ADASIM are functional simulators using C and Ada,[1] respectively. The AIVD tool set creates a loosely-coupled interface with the ADAS simulators through the generation of ADAS script files.

For more information on the ADAS simulators, refer to the related documents referenced in Section 1.3 of this manual.

2.2.5 The Allocator of Software to Hardware (ASH)

The Allocator of Software to Hardware (ASH) has been enhanced in the AIVD program to employ simulated annealing methodology in assigning software functions to hardware resources. These functions and resources are represented in ADAS software graphs and hardware graphs, respectively. The allocations are made in an effort to minimize the maximum utilizations of given hardware resources.

2.3 The AIVD Data Bases

2.3.1 The Application Domain Hierarchy

The Application Domain Hierarchy (ADH) organizes the descriptions of generic algorithms by application. Each level of the hierarchy will have a template file associated with it containing the templates for all the lower level algorithms. With each level may be associated transformation rules that can be applied to any of the graphs in the lower-levels of the ADH. At the lowest level, there will be several types of information:

- A set of node and arc templates which refer to common libraries of basic algorithms, such as sort algorithms or Fourier transform algorithms.
- The ADL programs for each of the algorithms which describe the performance characteristics of the algorithm, typically in terms of the number of instructions executed or in terms of the number of operations required.
- Ada program fragments for the primitive algorithms which are developed to the level needed to support functional simulation of the algorithm.

[1] Ada is a registered trademark of the U.S. Government.

2.3.2 The Hardware Domain Hierarchy

The Hardware Domain Hierarchy (HDH) contains the descriptions of potential architectures. For each architecture, there are several types of information:

- A set of node and arc templates referring to common libraries of components, such as 1750A or 32020 processors, PI-Bus or TMDE interconnections, and different memory architectures.
- A set of transformation rules for building an instance of the architecture from the components.
- A set of transformation rules for adapting a software algorithm to the architecture.
- A set of ADL descriptions of the performance characteristics of the components of the architecture and information about the operating system for the architecture, such as compilation rates for estimating machine code size from source code.

2.3.3 The Transformation Rule Base

The transformation rule base is a set of ADAS graphs, each of which describes the patterns and transforms of a consistent set of rules designed to transform a software data flow graph in order to achieve a specific purpose. Transformation rule bases may be developed for a specific application by the user, or may be constructed by selecting and then merging rules that are distributed throughout the ADH and the HDH.

2.3.4 Attribute Definition Language Programs

Attribute Definition Language (ADL) programs describe the ADAS node, graph, and arc attributes in terms of formulas which use system parameters defined by the user. These parameters typically consist of mission parameters, hardware performance parameters, and parameters used for back-annotation.

The ADL supports two types of communication: *inheritance* and *synthesis*. Inheritance allows parameters to be defined at a global level (such as the root graph of the system software hierarchy) and inherited into all nodes and arcs in the subgraph below the graph where they are defined. Inheritance is appropriate for distributing mission parameters to all the software nodes. Synthesis allows parameters to be computed by aggregating the values of ADAS attributes in a subgraph to define the value of an attribute in the parent node or graph. Synthesis is appropriate for back-annotating either software or hardware graph hierarchies.

2.3.5 ADAS Software Data Base

The ADAS software data base is a hierarchy of data flow graphs. It serves as the central working data base during an AIVD session, which will typically involve multiple trade-off experiments. At the top levels of the hierarchy, it describes the functions of the system. At the middle levels of the hierarchy, it describes the algorithms and data structures that provide the functions required of the system. At the lowest levels of the hierarchy, it describes the partitioning of the algorithms and data structures to fit onto the system hardware architecture.

The attributes of the ADAS software graph hierarchy describe the performance characteristics of the system, including the amount of time it takes to perform each atomic action of each algorithm, and how much input and output data is required for each atomic action.

2.3.6 ADAS Hardware Data Base

The ADAS hardware data base is a hierarchy of graphs which describe the structure and components of the system hardware architecture.

The ADAS hardware graph hierarchy constrains the possible mappings performed by ASH. Each component of the architecture belongs to a *module_class* and each atomic action of each algorithm must be mapped to a hardware component with the same *module_class*. Each arc of the software data flow graph must map onto a node or arc of the hardware graph. During simulation, atomic actions of algorithms must contend for the shared resources of the hardware components.

The attributes of the ADAS hardware graph hierarchy describe the performance characteristics of the components of the architecture, such as sensors, processors, memories, and interconnections.

3. AIVD METHODOLOGY

AIVD was designed to support a system design methodology which has been used by RTI and by ADAS customers. The nature of the design process and in particular the nature of software/hardware codesign is described below.

3.1 System Design Process Characteristics

The types of system design problems that ADAS was developed to support share several common characteristics:

Iterative Processing System design is an iterative process. There are several types of cycles that can occur in the design process, and a system design tool needs to support the process of working through all of those cycles efficiently. Typical types of iterations are:

- iterating the incremental design of software and hardware
- iterating the incremental design of different levels of the hierarchy
- iterating the definition of models based on different system parameters

The most important iterative cycle to support efficiently is the innermost iteration, i.e., the cycle which is repeated most often. AIVD focuses on supporting the incremental modifications to the model based on different system parameters that are inherent in performing trade-off studies. This gives the system architect more time to focus on designing different variations of the software and the hardware, rather than building the models required to perform trade-off analyses.

Quantitative Trade-offs ADAS was designed to assist the system architect by providing quantitive performance information. AIVD extends that support to include the many performance analysis required to make trade-offs. Thus, critical issues for a system architect using AIVD is defining the set of trade-off experiments to be performed, defining the parameters for those experiments, and deciding how to interpret the results of the simulations.

Software and Hardware Interactions A basic tenet of the ADAS methodology is that system performance is based on the interaction between the software and the hardware. ADAS models this interaction in terms of the mapping of software to hardware and the contention of software processes for shared hardware resources.

Multidimensional Design Space The need to perform trade-off studies across many different design options leads to a multidimensional design space. Typical trade-off studies must consider a bewildering number of different design options and system parameters. Each option and parameter defines a new dimension of the design space. Each simulation evaluates one point in that design space. Thus, the system architect must use the modeling resources wisely if a large number of options and parameter values is to be considered. AIVD is designed to allow the system architecture to explore a wider range of options.

In AIVD, the cycle that was chosen is the process of evaluating points in a large, parameterized design space.

3.2 The ADAS View of Software/Hardware Codesign

The first basic assumption that ADAS makes it that timing is a critical design issue. ADAS focuses on the design and assessment of real-time computer systems. A *real-time system* is a system in which there are critical requirements upon the time at which events occur. These requirements may take several forms:

- The rate at which an input data stream is consumed and processed without losing data may be critical. For example of this type of requirement is that a sensor has to be sampled at a particular rate.
- The rate at which an output data stream is produced may be critical. For example of this type of requirement is that an image has to be outputted 30 times a second in order to be flicker free.
- The sequence of events may be critical. For example, a parachute must be deployed before the landing gear is extended.
- The time between events may be critical. For example, the time between sighting a target and launching a weapon must be less than 5 seconds.

The second basic assumption that ADAS makes is that the system software can be modeled with hierarchical data flow graphs. This assumption is based on extensive work on the use of data flow graphs for structured systems analysis and structured system design which has been pioneered by Tom DeMarco.[1] ADAS extends this model

[1]Tom DeMarco. *Structured Analysis and System Specification.* Englewood Cliffs; Yourdon Press, 1979.

by providing attributes for graphs, nodes, and arcs which describe the performance characteristics of the system. These attributes lead to a capability to simulate the performance of the system using a form of Petri Nets called *marked graphs.*

The third basic assumption that ADAS makes is that the system design must take into account the interactions between software and hardware. ADAS has modified the marked graph models in order to account for the contention for hardware resources that are experienced when independent software processes share hardware resources.

The ADAS methodology is based on performing a series of steps to build a model of the system and then evaluate its performance:

- Building hierarchical data flow diagrams which describe the system software.
- Defining the performance attributes of the system software.
- Building hierarchical block diagrams which describe the structure of the system hardware.
- Mapping software to hardware.
- Simulating performance at the marked graph level.
- Evaluating the performance results produced by the simulation.

ADAS provides mapping and simulation tools and a graphical user interface to support the construction of models. AIVD focuses on assisting the user in iteratively making incremental changes in the models in order to evaluate many trade-off options.

3.3 The AIVD Design Process

The AIVD design process is shown in Figure 3.1.

Select Architecture from Hardware Domain Hierarchy. The architecture is described by:

- A set of component models (e.g., CPUs, IOPs, Memories, Buses).
- A set of basic models for the interconnection patterns (e.g., a hypercube would have a 2 node hypercube as a basic component.).
- A set of rewrite rules which would allow particular instances of the architecture to be built from the basic models (e.g., a hypercube would have a rewrite rule for building a larger hypercube from a smaller one by copying the existing hypercube and then connecting all the corresponding nodes in the two copies).

- A set of attribute definitions which would be used to configure the algorithms.

Transform Architecture In order to simulate a system, a specific instance of the architecture must be constructed. This is done by configuring an architecture using the proper number of each type of component, and connecting them according to the connection patterns of the architecture. With AIVD, the system architect uses the graph transformation system to connect the components according to the specification for the architecture as encapsulated in the graph transformation rules for the architecture. A design trade-off will often experiment with different numbers of components. Figure 3.1 shows the change in these numbers, *Network Size Parameters* being the middle loop for the design process.

Modify Architecture Attributes Typical architecture attributes are processor speed, memory bandwidth, interconnect bandwidth, and memory size. These attributes will affect the mapping process and the algorithm node processing times. A design trade-off will often require experiments with different values for these attributes.

Select Algorithm The user selects appropriate algorithms for each of the system functions from Application Domain Hierarchy. The ADH is organized hierarchically by function in order to make it easier for the user to determine which algorithms in the library are appropriate for the particular function and application. Furthermore, the ADH may have several variations on a particular algorithm, where different variations are associated with different hardware architectures. Thus, the user may use information about the architecture, as provided by attribute definitions, to select appropriate variations on the algorithms suited to the functions of the application.

Transform Algorithm The next step is to customize each of the algorithms in the system to fit the available architecture resources. This is done in a global context, so that trade-offs for resources can be made between algorithms for different functions of the system. When the algorithms are selected from the ADH, they come equipped with different transformations which are needed to adapt the algorithm to different instances of architectures. Once the architecture has been defined, its characteristics can be used to select the appropriate transformations for the algorithms. The architect uses the graph transformation system to customize the algorithm. Typical reasons for making architecture-specific transformations are:

- Increase the parallelism of an algorithm.
- Decrease the parallelism of an algorithm.

- Model the communication costs of an algorithm as distributed across the architecture.
- Provide the data or processing redundancy needed to support system fault tolerance.

Modify Algorithm Attributes Once the algorithm has been transformed, the algorithm attributes such as data structure size and operation counts are combined with the architecture attributes such as processor speed, memory bandwidth, and interconnect bandwidth to define the produce and consume rates and firing delays of the performance model. This process is performed by ADL-EVAL, using the attribute definition programs attached to the algorithm nodes, arcs, and graphs, and the ADL include files provided by the architecture model.

Map Software to Hardware The fully instantiated algorithm is mapped to the fully instantiated architecture using the Allocator of Software to Hardware.

Simulate System In AIVD, the user can evaluate performance by simulating the system at the marked graph level. Alternatively, the user can build a functional model using C or Ada code segments for each leaf node, and then perform a combined functional and performance simulation.

Evaluate Simulation Results This critical step is performed manually by the AIVD user, who must decide whether or not to continue the search through the design space, and which point in the design space should be evaluated next. The user implements the decision by setting new parameter values in ADL files, and by invoking the appropriate tool to start the next cycle through the process.

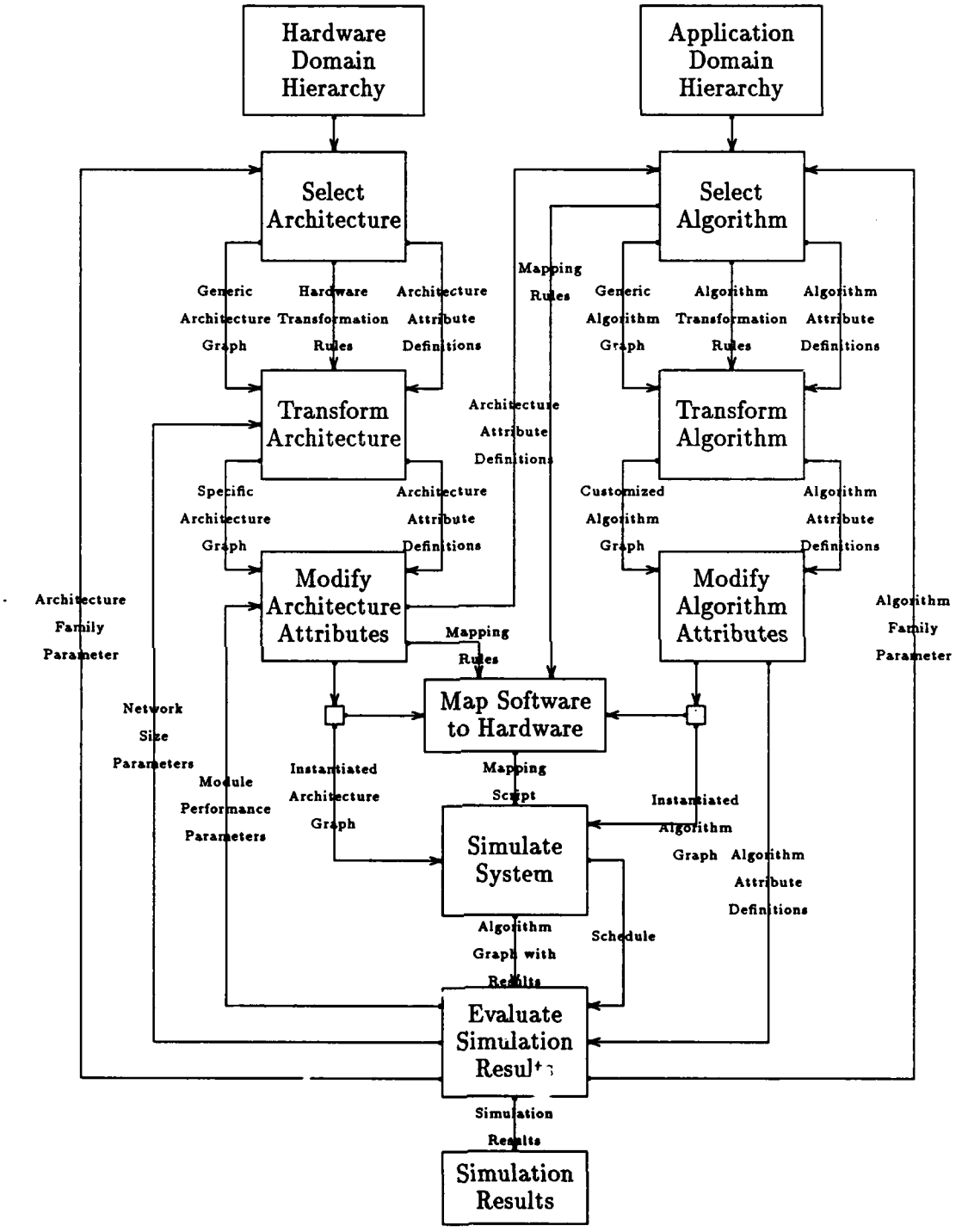

Figure 3.1: The AIVD Design Process

4. INTELLIGENT USER INTERFACE

4.1 Overview

The AIVD Intelligent User Interface (IUI) is an enhanced version of the Architecture Design and Assessment System (ADAS) Version 2.5 graph editor EDIGRAF. It is executed by entering the command

iui *dbase* **-s** *script* **-d** *gterm*

where:

- *dbase* is the name of the graph data base,
- *script* is the name of a script file, and
- *gterm* is the name of the graphics display device.

4.2 IUI Inputs

4.2.1 ADAS Graph Files

The *node_user_text* attribute of each node contains the name of the associated help file. Similarly, the *arc_user_text* attribute of each arc contains the name of the associated arc help file, and *graph_user_text* contains the name of the associated graph help file. The *node_user_file_name* attribute of each node contains the name of the associated Attribute Definition Language (ADL) file. Similarly, the *arc_user_file_name* attribute of each arc contains the name of the associated arc ADL file, and *graph_user_file_name* contains the name of the associated graph ADL file.

4.2.2 ADAS Template Files

The IUI keeps track not only of the current template file (i.e., the template file associated with the current graph), but also of template files associated with the graphs in the current graph hierarchy.

4.2.3 New Commands

IUI extends the current set of ADAS EDIGRAF commands with four types of new commands:

- **help** commands, which display files associated with graphs, nodes, arcs, node templates, and arc templates to help the user.
- **view** commands, which display for the user the structure of the graph and template hierarchies.
- **edit text** commands, which allow the user to edit text files associated with graphs, nodes, arcs, node templates, and arc templates.
- **subtmpl** command, which allow the user to edit the graph specified by the *subgraf_file_name* of a node template.

4.2.3.1 Help Commands

The **help** commands display help files about the current graph, nodes, and arcs, and about the current node and arc templates.

help graph

The file whose name is the value of the *graph_user_text* attribute of the current graph is displayed on the terminal screen using the VMS **type/page** command.

help node *node_id*

A menu of the names of the nodes of the current graph is displayed. When a node is selected (either by picking off the display, picking off the menu, or typing in the node name), then the file whose name is the value of the *node_user_text* attribute of the selected node is displayed on the terminal screen using the VMS **type/page** command.

help nodetmpl *template*

A menu of the current set of node templates is displayed. When a node template is selected (either by picking off the menu or by typing in the template name), then the file whose name is the value of the *node_user_text* attribute of the selected node template is displayed on the terminal screen using the VMS **type/page** command.

help arc *node_id output_port_number*

A menu of the names of the nodes of the current graph is displayed. When a node is selected (either by picking off the display, picking off the menu, or typing in the node name), then a menu of output port numbers for the selected node is displayed. When an output port is selected (either by picking off the menu or by typing in the port number), then the file whose name is the value of the *arc_user_text* attribute for the selected arc is displayed on the terminal screen using the VMS **type/page** command.

help arctmpl *template*

A menu for the current set of arc templates is displayed. When an arc template is selected (either by picking off the menu or by typing in the template name), then the file whose name is the value of the *arc_user_text* attribute for the selected arc template is displayed on the terminal screen using the VMS **type/page** command.

4.2.3.2 View Commands

The **view** commands give the user a sense of the template and subgraph hierarchies associated with the current graph. The template hierarchies represent the Application Domain Hierarchy (ADH) available below the current graph, i.e., all possible domain selections. Recall that the application domain hierarchy consists of a hierarchy of empty graphs, each with its own template file, help file, and associated ADL files. This view capability allows the user, for example, to determine which primitive functions are available at the lowest levels of the current application domain hierarchy. The subgraph hierarchies give him a sense of where certain functions are performed in the system. For example, he might like to know which of his top-level functions requires an FFT operation, or in which subgraph the edge detection function is performed.

view hierarchy template [**name** | **graph_name** | **file_name** | **full_file**] *level*

This command displays the application domain hierarchy, as defined by the hierarchy of one of the following user-specified options: template **names**, template subgraph **graph_names**, template subgraph **file_names**, or the template subgraph **full** path **file** names. The top level of the hierarchy consists of the user-specified option of the current graph. This command traverses the current graph hierarchy and collects the user-specified option information of all the templates. This information is presented in a hierarchical fashion, with the

hierarchy represented by indentation. The printing on the terminal screen stops at page boundaries, as in the VMS **type/page** command. If *level* = 0, then all levels are printed. If *level* = $i > 0$, then only the first i levels of the hierarchy are presented by the **view** command. No negative values of *level* are permitted.

For example, if the **name** option is selected by the user, T is the name of a node template at level n, the node class of T is *internal*, and T has a specified subgraph that exists and has a template file name F specified, then the name of any node template S in F is displayed at level $n + 1$. A problem can result from circular definitions of the hierarchy. If node template T in template file F has subgraph G and the template file for G is also F, then a circular definition of the hierarchy will result. In this situation, only the highest level is displayed.

view hierarchy graph [**name** / **graph_name** / **file_name** / **full_file**] *level*

This command displays lower levels of the current software graph (i.e., application domain hierarchy) as defined by the hierarchy of one of the following user-specified options: node **names**, node subgraph **graph_names**, node subgraph **file_names**, or the node subgraph **full** path **file** names. This command traverses the current graph hierarchy and collects the user-specified option information on all the nodes. This information is presented in a hierarchical fashion, with the hierarchy represented by indentation. The printing on the terminal screen stops at page boundaries, as in the VMS **type/page** command. If *level* = 0, then all levels are printed. If *level* = $i > 0$, then only the first i levels of the hierarchy are presented by the **view** command. No negative values of *level* are permitted.

4.2.3.3 Edit Commands

These **edit** commands were created to allow the edit of a text file referenced by an attribute without having to leave the EDIGRAF session or retype the name of the file.

edit text graph [*graph_user_text* / *graph_user_file_name*]

This command edits a text file associated with two attributes associated with graphs: *graph_user_text* for help files and *graph_user_file_name* for ADL files.

When this command is entered, EDIGRAF calls the appropriate editor (EDT on VMS) and passes the attribute value to the editor as the name of the file to be edited.

edit text [**node** / **nodetmpl**] [*node_user_text* / *node_user_file_name*]

This command edits a text file associated with two name attributes associated with nodes or node templates: *node_user_text* for help files and *node_user_file_name* for ADL files. When this command is entered, EDIGRAF calls the appropriate editor (EDT on VMS) and passes the attribute value to the editor as the name of the file to be edited.

edit text [**arc** / **arctmpl**] [*arc_user_text* / *arc_user_file_name*]

This command edits a text file associated with two attributes associated with arcs or arc templates: *arc_user_text* for help files and *arc_user_file_name* for ADL files. When this command is entered, EDIGRAF calls the appropriate editor (EDT on VMS) and passes the attribute value to the editor as the name of the file to be edited.

4.2.3.4 Subtmpl Command

A **subtmpl** *node_template* command was created to allow one to use EDIGRAF on a graph referenced by the *subgraf_file_name* attribute of a node template, without having to leave the current EDIGRAF session or retype the name of the file. This is similar to the present **subgraf** command in EDIGRAF except it enters an EDIGRAF session with a node template's *subgraf_file_name* instead of a node's *subgraf_file_name.* A **quit** command returns the user to the previous graph.

For example, assume the graph being used in an IUI session is `x.swg`, its *template_file* is `x.swt`, `tempa` is the name of a node template in `x.swt`, and the *subgraf_file_name* attribute for `tempa` is `y.swg`. When the command **subtmpl** `tempa` is entered the user will be editing `y.swg`. A **quit** command returns the user to the `x.swg` IUI session.

4.3 IUI Outputs

As an example of IUI output, there is a graph radar.swg which has two node templates, `fft` and `armult`. Its full path name is `project:[aivd.adh.radar]radar.swg`. From

these two node templates two nodes are created, `fft0` and `armult0`. Both the nodes and the node templates from which they were created share the same *graph_names* and *subgraf_file_names*. These nodes and node templates comprise *level* 1 of the radar template and graph hierarchies. The graph fft.swg has four node templates, `mult`, `add`, `read` and `write`. From these four templates four nodes are created: `mult0`, `add0`, `read0` and `write0`. Both the nodes and the node templates from which they were created share the same *graph_names* and *subgraf_file_names*. These nodes and node templates comprise *level* 2 of the radar template and graph hierarchies.

If, during an IUI session on radar.swg, **view hierarchy template name 0** is entered, the resulting **name** hierarchy is

```
fft
    mult
    add
    read
    write
armult,
```

while if **view hierarchy graph name 0** is entered, the resulting **name** hierarchy is

```
fft0
    mult0
    add0
    read0
    write0
armult0.
```

If **view hierarchy template graph_name 0** is entered, the resulting **graph_name** hierarchy is

```
Fast_Fourier_Transform
    Multiply
    Add
    Memory_Read
    Memory_Write
Array_Multiply,
```

while if **view hierarchy graph file_name 0** is entered, the resulting **file_name** hierarchy is

```
[.fft]fft.swg
    [.mult]mult.swg
    [.add]add.swg
    [.read]read.swg
    [.write]write.swg
[.armult]armult.swg.
```

Finally, if **view hierarchy graph full_file 0** is entered, the resulting **full_file** hierarchy is

```
project:[aivd.adh.radar.fft]fft.swg
    project:[aivd.adh.radar.fft.mult]mult.swg
    project:[aivd.adh.radar.fft.add]add.swg
    project:[aivd.adh.radar.fft.read]read.swg
    project:[aivd.adh.radar.fft.write]write.swg
project:[aivd.adh.radar.armult]armult.swg
```

5. ATTRIBUTE DEFINITION LANGUAGE

5.1 Overview

The Attribute Definition Language (ADL) provides a flexible interface for defining ADAS component attributes in terms of functions and expressions of ADL variables. A noninteractive tool, ADLEVAL reads an ADAS graph hierarchy and its associated ADL files and generates a script file which rewrites the graph hierarchy's attributes based on the rules defined in the ADL files. The modified graph can then be transformed, mapped, simulated, and analyzed by the other tools in the ADAS set. A second program, ADLLINT, can be used to check the syntactic correctness of individual ADL programs.

Attributes and variables can be assigned values based on values that are *inherited* from the parent graph level or *synthesized* from values in the subgraph or the current graph. ADLEVAL processes ADL programs in two passes: it processes inherited variables top-down on the first pass and synthesized variables bottom-up on the second pass.

5.2 The Attribute Definition Language

5.2.1 ADL Data Types and Constants

ADL supports the ADAS attribute data types described in Table 5.1. Each of the data types may have corresponding constants in the language. Integer constants are always in decimal format, and may have an optional sign in front of them. The signs on the exponent of a floating point number are optional. Strings can have any ASCII characters except double quotes.

Type	*Description*	*Examples*
float	Floating point number	`2.375,-12.11e-10`
integer	Integer number	`-4, 22`
boolean	Boolean value	`true, false`
string	Text string (up to 200 chars.)	`"This_Is_A_Sample_String"`

Table 5.1: ADAS Attribute Data Types

5.2.2 ADAS Attributes

ADL programs read ADAS attribute values from the current graph and write attribute values into script files. The use of ADAS attributes is described in Table 5.2 through Table 5.6.

An ADL program can only overwrite attributes whose modification *status* is **M** (modifiable) or **P** (program modifiable). An attempt to redefine the value of an attribute whose modification status is **N** (not modifiable) will cause ADLEVAL to print an error message.

Only graph ADL programs may reference ADAS graph attributes or assign values to them. The name, type, and use of ADAS graph attributes are described in Table 5.2.

Attribute Name	Data Type	ADLEVAL Access
graph_version_number	string	read and write
time_unit	string	read only
conversion_factor	string	read only
graph_user_float	float	read and write
graph_user_integer	integer	read and write

Table 5.2: ADAS Graph Attributes Used By ADLEVAL

Only node ADL programs may assign values to ADAS node attributes. Node ADL programs may use ADAS node attributes in inheritance statements. Both node and graph ADL programs may reference ADAS node attributes in synthesis statements. The name, type, and use of ADAS node attributes are described in Table 5.3.

Only node ADL programs may assign values to ADAS inport and outport attributes. Node ADL programs may use ADAS inport and outport attributes in inheritance statements. The name, type, and use of ADAS inport and outport attributes are described in Table 5.4 and Table 5.5.

ADAS port attributes are referred to in an ADL expression by entering their ADAS attribute name and an extension selecting an individual inport or outport. Port extensions are of the form **(in*i*)** or **(out*i*)**, where *i* is the port number (from 0 to 63). Figure 5.8 shows the format for referencing input and output ports.

Only arc ADL programs may assign values to ADAS arc attributes. Arc ADL programs may use ADAS arc attributes in inheritance statements. The name, type, and use of ADAS node attributes are described in Table 5.6.

Attribute Name	Data Type	ADLEVAL Access
node_color	string	read and write
hw_module	string	read and write
execution_order	integer	read and write
node_utilization	float	read and write
module_utilization	float	read and write
firing_delay	float	read and write
node_latency	float	read and write
times_fired	integer	read and write
when_next_available	float	read only
module_class	string	read and write
node_orientation	string	read and write
node_width	float	read and write
node_height	float	read and write
node_user_float	float	read and write
node_user_integer	integer	read and write
trace_flag	integer	read and write

Table 5.3: ADAS Node Attributes Used By ADLEVAL

Attribute Name	Data Type	ADLEVAL Access
in_token_data_type	string	read only
token_consume_rate	integer	read and write
firing_threshold	integer	read and write
initial_token_count	integer	read and write

Table 5.4: ADAS Input Port Attributes Used By ADLEVAL

Attribute Name	Data Type	ADLEVAL Access
out_token_data_type	string	read only
token_produce_rate	integer	read and write

Table 5.5: ADAS Output Port Attributes Used By ADLEVAL

Attribute Name	Data Type	ADLEVAL Access
arc_color	string	read and write
queue_size	integer	read and write
token_data_type	string	read only
average_token_count	float	read only
maximum_token_count	integer	read only
current_token_count	integer	read only
token_access_count	integer	read only
arc_user_float	float	read and write
arc_user_integer	integer	read and write

Table 5.6: ADAS Arc Attributes Used By ADLEVAL

5.2.3 ADL Variables

ADL variable names may contain alphanumeric characters or the underscore (_), and they must begin with an upper or lower case letter of the alphabet. A number of ADL names are reserved; they are listed in Table 5.7. All ADAS graph, node, arc, inport, and outport attribute names are also reserved.

and	arc	avg	—	bool	count	end
endfor	endif	false	float	for	graph	if
include	inport	int	log2	max	—	min
—	node	not	or	outport	parent	select
sqrt	string	sum	—	synth	synthesis	synthesize
then	true	var	—	where	—	—

Table 5.7: ADL Reserved Words

5.2.3.1 Local Variables

Local variables may be declared in graph, node, or arc ADL programs and be assigned values based on the values of constants, attributes, and variables. The type of a local variable is defined by its declaration; the value of a local variable is defined by an inheritance assignment statement in the same ADL program where it is declared. Local variables declared in a graph ADL file can also be referenced by graph, node, or arc ADL files in the subgraphs. Local variables declared in a node ADL file can be referenced by the graph ADL file for the node's subgraph.

5.2.3.2 Graph Variables

A graph, node, or arc ADL program can reference local, graph, or parent variables declared in the parent graph's ADL program by declaring such variables as *graph variables.* A graph, node, or arc ADL program cannot assign values to a graph variable. The names used in the parent graph ADL program must match the names used in the graph, node, or arc ADL program. The data type of a graph variable is determined by the type used in the parent graph.

5.2.3.3 Parent Variables

A graph ADL program can reference local, graph, or parent variables declared in the parent node's ADL program by declaring such variables as *parent variables.* A graph ADL program cannot assign values to a parent variable. The names used in the parent node ADL program must match the names used in the graph ADL program. The data type of a parent variable is determined by the type used in the parent node.

5.2.4 ADL Expressions

An ADL expression is either a constant, an ADAS attribute, a variable, or is constructed by using an operator to combine one or more ADL expressions. ADL expression values can be assigned to variables or ADAS attributes. ADL supports inherited and synthesized expressions. The order of evaluation of expressions depends on operator precedence as described in the sections below. Left and right parentheses can be used to force a certain order of evaluation or make the order of precedence explicit; expressions in parentheses are evaluated first, starting with the deepest embedded parentheses.

5.2.4.1 Math Operators

A number of mathematical operators are provided for use in ADL expressions; they include operators for addition, subtraction, multiplication, and division. Multiplication, division, addition, and subtraction all operate on either integers or floating point data types. Table 5.8 lists these operators in order of precedence from highest (top) to lowest (bottom). Operators in the same row have the same precedence; they are evaluated in an expression from right to left.

Operators	*Meanings*
*,/	Multiplication, Division
+,-	Addition, Subtraction

Table 5.8: ADL Mathematical Operators

5.2.4.2 Logical Operators

Conditional ADL expressions use a set of relational and logical operators to qualify ADAS attribute and ADL variable evaluation. ADL provides a logically sufficient set of logical operators. Table 5.9 lists these operators in order of preference from highest (top) to lowest (bottom). Operators with the same precedence are evaluated in an expression from right to left.

Operators	*Meanings*	*Example*
and	intersection	`(A > 5) and (A < 10)`
or	union	

Table 5.9: ADL Logical Operators

5.2.4.3 Relational Operators

ADL provides relational operators for equality, inequality, and range tests. Table 5.10 lists these operators, which can be applied to either integer or floating point values. A subset of these operators can be applied to string and boolean values. (All of these operators have the same precedence.) Operators with the same precedence are evaluated in an expression from right to left.

5.2.4.4 String Operators

ADL includes a concatenation operator and two relational operators for strings as indicated by Table 5.11.

5.2.4.5 Aggregation Functions

ADL provides a number of aggregation functions that may be included in synthesized expressions. They return an **integer** or **float** value calculated or composed from a

property of a cluster of components in the subgraph or the current graph. These functions are listed in Table 5.12.

Operators	*Meanings*	*Example*
==, /=	Equality, inequality	`Name /= 'Filter'`
<, >	Less than, greater than	`Delay < 5.0`
<=, >=	Less than or equal to, greater than or equal to	`token_consume_rate(in0) <= 10`

Table 5.10: Relational Operators

Operator	*Meaning*	*Example*
+	concatenation	`"CPU" + "0"`
==	equality test	`hw_module =="CPU"`
/=	inequality test	`hw_module /= "DPU"`

Table 5.11: ADL String Operators

Name	*Returns*	*Description*	*Example*
max	integer, float	Maximum	`max(queue_size`
sum	integer, float	Sum	`sum(node_user_integer`

Table 5.12: ADL Aggregation Functions

5.2.5 ADL Statements

5.2.5.1 Comments

Comments contain a double dash ('--') at the beginning of a line or immediately after an ADL statement on the same line followed by zero or more ASCII characters. Comments are terminated by new lines. They can be used anywhere in an ADL program.

5.2.5.2 Declaration Statements

A variable that is inherited by a node or arc ADL program from the current graph ADL program must be declared as a *graph variable* in the node or arc ADL program. Similarly, a variable that is inherited by a graph ADL program from the parent graph ADL program must be declared in the ADL program for the subgraph. Graph variables inherit both their type and their value from the parent graph ADL program variable, which must have the same name. Graph variables cannot occur on the left hand side of an assignment statement. Graph variables are declared either local, parent, or graph in the parent graph ADL program. Graph variables which are declared in graph ADL programs can be referenced as graph variables in the node or arc ADL programs for the current graph and in the graph ADL programs for all immediate subgraphs. Graph variables which are declared in a node ADL program can be referenced as parent variables in the graph ADL program for an immediate subgraph. Graph declarations take the following form:

graph : <name>{,<name>}*;

where <name> is a graph variable's name. Note that multiple variables can be declared with a single declaration.

A variable that is inherited by a graph ADL program from a parent node's ADL program must be declared as a *parent variable* in the subgraph's ADL program. Parent variables inherit both their type and their value from the parent node ADL program variable, which must have the same name. Parent variables cannot occur on the left hand side of an assignment statement. Parent variables are declared either local or graph in the parent node ADL program. Parent variables can be referenced as graph variables in the node or arc ADL programs for the current graph and in the graph ADL programs for all immediate subgraphs. Parent declarations take the following form:

parent : <name>{,<name>}*;

where <name> is a variable's name. Note that multiple variables can be declared with a single declaration.

A variable that is assigned a value generated by an inheritance expression must be declared as a *local variable* in a graph, node, or arc ADL program. The declaration of a local variable defines the type of the variable. Local variables are assigned their values through the use of an inheritance assignment statement, where the expression on the right hand side is an inheritance expression. Local variables which are declared in graph ADL programs can be referenced as graph variables in the node or arc ADL

programs for the current graph and in the graph ADL programs for all immediate subgraphs. Local variables which are declared in node ADL programs can be referenced as parent variables in the graph ADL program for an immediate subgraph. Local declarations take the following form:

<type> : <name>{,<name>}*;

where <type> is the type of the variables and <name> is a variable's name. Note that multiple variables can be declared with a single declaration.

5.2.5.3 Assignment Statements

An assignment statement assigns a value to an attribute or variable in a graph's ADL file in terms of values that are inherited from the parent node's or parent graph's ADL file. Assignment statements take the form:

{<attribute> | <name>} = <expr>;

where <attribute> is an ADAS attribute, <name> is the name of a local variable, and <expr> is an ADL expression. If an expression contains only local, graph, or parent variables, then the expression is an **inheritance expression** and can be used in inheritance assignment statements. If an expression contains aggregation functions, then it is a **synthesis expression** and must be used in a synthesis assignment statement. Synthesized expressions can contain aggregate function calls and local, or synthesized variables. Synthesized expressions can contain inherited variables since inheritance precedes synthesis during evaluation.

```
--
-- Graph ADL for filter.swg: Initialize global variables
--
graph : mips;
parent : array_size;
int : cutoff;

cutoff = array_size/mips;
```

Figure 5.1: Filter Graph ADL Program

5.2.6 ADL Programs

An ADL file contains one or more lines of ASCII characters that define variables, variable values, and attribute values. A section containing variable *declaration statements* precedes one or more sections containing attribute and variable *assignment statements*. Any declaration or assignment section may be empty. Individual statements are separated by semicolons. Within any section, statements are evaluated according to their order of appearance in the file. ADL is a free-format language; individual words (i.e., variable names, keywords, attribute names, operators) can be separated by blanks or new lines.

5.2.6.1 Graph ADL Program Structure

A graph ADL program contains the following subsections:

{<graph inherited variable declaration>}*
{<parent node inherited variable declaration>}*
{<local variable declaration>}*
{<inheritance statements>}*
{<synthesis statements>}*

5.2.6.2 Node ADL Program Structure

A node ADL program contains the following subsections:

<graph inherited variable declaration>*
<local variable declaration>*
<inheritance statements>*
<synthesis statements>*

5.2.6.3 Arc ADL Program Structure

An arc ADL program contains the following subsections:

<graph inherited variable declaration>*
<local variable declaration>*
<inheritance statements>*

5.3 The ADLEVAL Program

ADLEVAL is invoked as a stand-alone tool while running Prolog. The command to start Prolog, **quintus_engine**, should also load the compiled ADLEVAL code, **adlrun**. During a Prolog session, the command **adleval** specifies the program, the parameter *filename* specifies the root ADAS graph, *debuglvl* is an integer (0-4, default = 1) that specifies the level of information to be displayed on the standard output during processing and *errhlt* is a boolean (default = 0) no halt determining whether the Prolog processing should halt if an error is encountered. The parameter *errhlt* should not be used without specifying *debuglvl*. Below is an example of a session invoking **adleval**. The bold font represents what the user enters, whereas the typewriter font represents the system's prompts and responses. All of the **adleval** commands are valid.

$ quintus_engine dua0:[aivd.src.adleval.july29]adlrun

```
Quintus Prolog Release 2.0 (VAX/VMS)
Copyright (C) 1987, Quintus Computer Systems, Inc. All rights reserved.
1310 Villa Street, Mountain View, California (415)965-7700
```

|?- **adleval** (‘*filename*’,*debuglvl*,*errhlt*).
OR
|?- **adleval** (‘*filename*’,*debuglvl*).
OR
|?- **adleval** (‘*filename*’).

```
LEXICAL ANALYSIS IN PROGRESS....

PARSING IN PROGRESS...

ADAS FILE filename HAS BEEN SUCCESSFULLY PARSED!

EVALUATING GRAPH ...
```

5.3.1 ADLEVAL Inputs

5.3.1.1 ADAS Data Base

ADLEVAL will read all graph files in the hierarchical ADAS data base. ADLEVAL works with the ADAS attributes described in Table 5.2 through Table 5.6.

5.3.1.2 ADL Files

ADLEVAL will examine every graph's *graph_user_file_name* to determine the name of the graph's ADL file. If the attribute has not been assigned a value, no ADL processing will take place for the graph. Note that no node in a graph without a graph ADL file can have an ADL program which references a graph variable.

ADLEVAL will examine every node's *node_user_file_name* to determine the name of the node's ADL file. If the attribute has not been assigned a value, no ADL processing will take place for the node. All node ADL files will be evaluated, including nodes whose *node_class* attributes are *internal*, *inport*, or *outport*. This means subgraphs will be expanded without flattening.

ADLEVAL will examine every arc's *arc_user_file_name* to determine the name of the arc's ADL file. If the attribute has not been assigned a value, no ADL processing will take place for the arc. The ADL files associated with arcs attached to graph port nodes will be evaluated.

5.3.2 ADLEVAL Processing

5.3.2.1 Processing Algorithm

ADLEVAL will evaluate ADL file entities using the following recursive algorithm:

```
Instantiate variables of graph ADL program
Process inherited section of graph ADL program
For each node in the graph
    Instantiate variables of node ADL program
    Process inherited section of node ADL program
    Invoke ADLEVAL on the node's subgraph (if any)
    Process synthesis section of node ADL program
For each arc in the graph
    Instantiate variables of arc ADL program
    Process inherited section of arc ADL program
Process synthesis section of graph ADL program
```

Figure 5.2: ADL Processing Algorithm

5.3.3 ADLEVAL Outputs

5.3.3.1 Script File

ADLEVAL will write a script file named *graph_name*.scr, where *graph_name* is the name of the top level graph, to set modified attribute values instead of modifying the ADAS data base files directly. The script file can be read during an EDIGRAF or GIPSIM session to set the attributes before simulation or display.

The script file will include commands for updating those ADAS attributes with write access listed in Table 5.2 through Table 5.6.

5.3.3.2 Errors

Two levels of error messages are generated by ADLEVAL:

- **serious error** In this case, an error message is generated and both syntax analysis and semantic processing continues at the next statement.
- **fatal error** All processing is stopped immediately.

The conditions in Table 5.13 will raise ADLEVAL errors.

Table 5.13: ADLEVAL Error Conditions

Error Condition	*Severity*
Nonexistent ADL file	Fatal
Nonexistent ADAS Graph file	Fatal
A duplicate variable declaration within an ADL file	Serious
A variable is declared as a reserved word	Serious
A parent declaration appearing in a top-level graph ADL file	Serious
A graph declaration appearing in a top-level graph ADL file	Serious
A parent declaration appearing in a node ADL file	Serious
A parent declaration appearing in an arc ADL file	Serious
A variable declared as graph in a graph ADL file is not declared in the parent graph's ADL file	Serious
A variable declared as parent in a graph ADL file is not declared in the parent node's ADL file	Serious
A variable declared as graph in an arc or node ADL file is not declared in the graph in which the node/arc is embedded	Serious
Referencing a variable which has not been declared	Serious
Referencing a variable which has not been defined	Serious
Assigning a value to an ADAS attribute that does not have an ADLEVAL access of write or has a modification status of not modifiable	Serious
Assigning a value to a parent or graph variable	Serious
Referencing an ADAS attribute which does not have ADLEVAL read access	Serious
Invalid floating point format	Serious
The operand of a mathematical operator is not of type integer or float	Serious
The operand of a logical operator is not of type boolean	Serious
The data type of an expression value does not match the data type of the variable or attribute being assigned	Serious

5.4 The ADLLINT Program

An ADL_LINT facility is provided for performing syntax checking on individual ADL files prior to executing ADLEVAL. ADL_LINT only detects syntax errors; it does not detect semantic errors and it does not generate a script file.

ADL_LINT is invoked as a stand-alone tool while running Prolog. The command to start Prolog, **quintus_engine**, should also load the compiled ADL_LINT code, **lintrun**. To check the syntax of a graph ADL file, the program **graph_ps** is invoked with the parameter *filename*, which specifies the name of the graph ADL file. Similarly, to check the syntax of a node ADL file, the program **node_ps** is invoked with the parameter *filename*, which specifies the name of the node ADL file. Finally, to check the syntax ot an arc ADL file, the program **arc_ps** is invoked with the parameter *filename* which specifies the name of the arc ADL file. Below is an example of a session invoking these three programs.

```
$ quintus_engine dua0:[aivd.src.adleval.july29]lintrun

Quintus Prolog Release 2.0 (Vax/VMS)
Copyright (C) 1987, Quintus Computer Systems, Inc. All rights reserved.
1310 Villa Street, Mountain View, California (415)965-7700

| ?- graph_ps('toplevel.adl').

Graph ADL file:  toplevel.adl

int :  DataArcValue , CtlArcValue , node_data ;

DataArcValue = 5 ;
CtlArcValue = 10 ;
node_data = 7 ;
graph_version_number = graph_version_number + time_unit ;
graph_user_float = graph_user_float + 1.5 + conversion_factor ;
graph_user_integer = graph_user_integer * 2 ;

yes
| ?- node_ps('subnode.adl').

Node ADL file:  subnode.adl

graph :  node_data ;
int :  consume0, consume1, produce0, parent_val ;
```

```
node_user_integer = node_user_integer + node_data ;
consume0 = token_consume_rate(in0) ;
>>> ''consume1 = token_consume_rate ( IN1 ) ; '' is an illegal
assignment statement.
produce0 = token_produce_rate(out0) ;
parent_val = 42 ;

synthesize
   node_user_integer = node_user_integer + max (node_user_integer);
   trace_flag = sum ( trace_flag ) ;
end synthesis

yes
| ?- node_ps('subnode.adl').

Node ADL file:  subnode.adl

graph :  node_data ;
int :  consume0 , consume1 , produce0 , parent_val ;

node_user_integer = node_user_integer + node_data ;
consume0 = token_consume_rate(in0) ;
consume1 = token_consume_rate(in1) ;
produce0 = token_produce_rate(in0) ;
parent_val = 42 ;

synthesize
   node_user_integer = node_user_integer + max (node_user_integer);
   trace_flag = sum ( trace_flag ) ;
end synthesis

yes
| ?- arc_ps('datarc.adl').

Arc ADL file:  datarc.adl

graph :  DataArcValue ;

arc_user_integer = arc_user_integer + DataArcValue ;
arc_color = ''orange'' ;
queue_size = queue_size + 5 ;
arc_user_float = arc_user_float + average_token_count ;
```

```
yes
| ?- halt

[ End of Prolog execution ]
```

5.5 Example

The following graphs and ADL code are an example of using ADLEVAL.

5.5.1 Top-Level Graph

Please refer to Figure 5.3, the *Top-Level ADAS Graph,* Figure 5.4, the *Top-Level ADAS Graph ADL File,* Figure 5.5, the *I/O Node ADL File,* Figure 5.6, the *Top-Level Internal Node ADL File,* Figure 5.7, the *Data Arc ADL File,* and Table 5.14, *ADLEVAL Results for Top-Level Graph.*

5.5.1.5 ADLEVAL Results

Table 5.14: ADLEVAL Results for Top-Level Graph

Component Name	*Attribute*	*Initial Value*	*After Inheritance*	*After Synthesis*
Graph file	*graph_user_integer*	—	—	107
Node Ain	*node_user_integer*	1	8	8
Node Bin	*node_user_integer*	2	9	9
Node Out	*node_user_integer*	4	11	11
Node Sub	*node_user_integer*	3	10	107
Arc post0	*arc_user_integer*	2	7	7
Arc pre0	*arc_user_integer*	1	6	6
Arc pre1	*arc_user_integer*	1	6	6

5.5.1.1 ADAS Graph

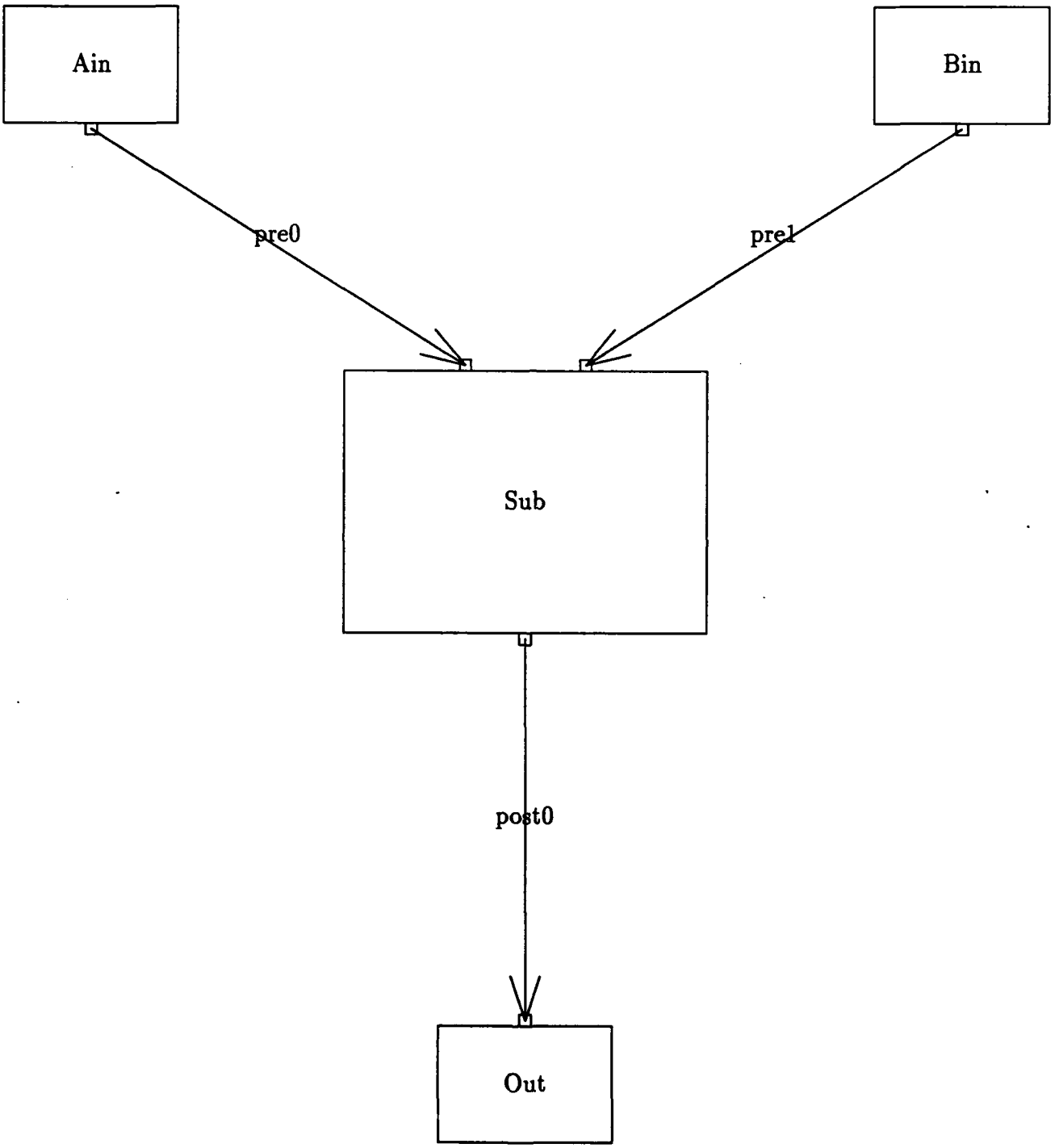

Figure 5.3: Top-Level ADAS Graph

5.5.1.2 Top-Level Graph ADL

```
--
-- Graph toplevel.swg: Initialize local variables and
-- store maximum node attribute value from current graph
--

int : DataArcValue, CtlArcValue;
int : node_data;

DataArcValue =  5;
CtlArcValue  = 10;
node_data    =  7;

synthesize
   graph_user_integer = max(node_user_integer);
end synthesis
```

Figure 5.4: Top-Level ADAS Graph ADL File

5.5.1.3 Top-Level Nodes ADL

```
--
-- Nodes Air, Bin, Out: set user attribute to
-- global value plus current attribute value
--

   graph : node_data;

   node_user_integer = node_user_integer + node_data;
```

Figure 5.5: I/O Node ADL File

```
--
-- Node Sub: set user attribute to global value plus
-- current attribute value plus maximum attribute value
-- from subgraph; place port attributes in variables for
-- inheritance
--

int : consume0, consume1;
int : produce0;
int : parent_val;

graph : node_data;

node_user_integer = node_user_integer + node_data;
consume0 = token_consume_rate(in0);
consume1 = token_consume_rate(in1);
produce0 = token_produce_rate(out0);
parent_val = 42;

synthesize
   node_user_integer = node_user_integer + max(node_user_integer);
end synthesis
```

Figure 5.6: Top-Level Internal Node ADL File

5.5.1.4 Top-Level Arcs ADL

```
--
-- Arcs of type pre, post, and data: set user attribute to
-- global value plus current attribute value
--

graph : DataArcValue;

arc_user_integer = arc_user_integer + DataArcValue;
```

Figure 5.7: Data Arc ADL File

5.5.2 Mid-Level Graph

Refer to Figure 5.8, the *Subgraph ADAS Graph,* Figure 5.9, the *Mid-Level Graph ADL File,* Figure 5.10, the *First Leaf Node ADL File,* Figure 5.11, the *Second Leaf Node ADL File,* Figure 5.12, the *Subgraph Internal Node ADL File,* Figure 5.13, the *Control Arc ADL File,* and Table 5.15, *ADLEVAL Results for Mid-Level Graph.*

5.5.2.1 ADAS Graph

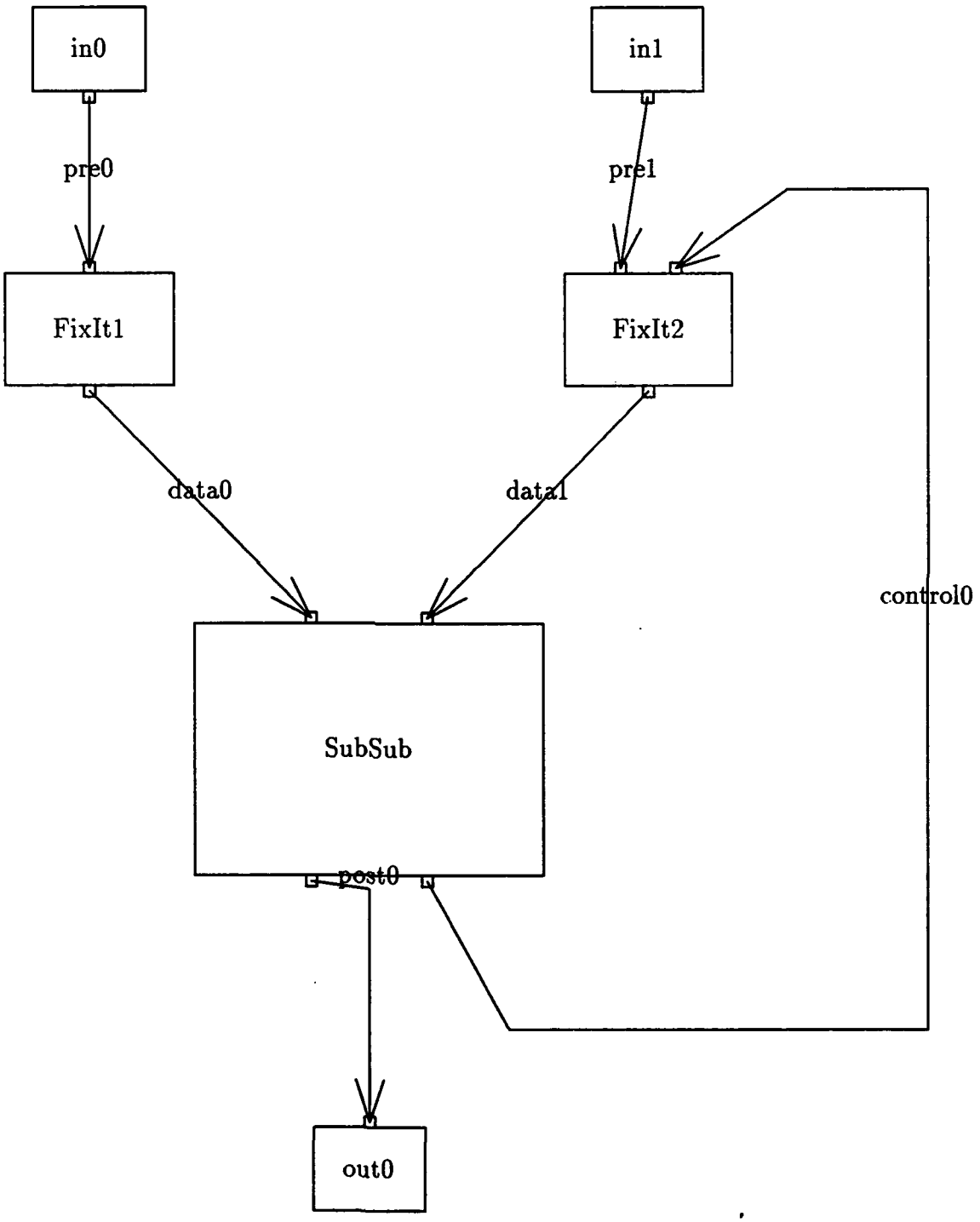

Figure 5.8: Subgraph ADAS Graph

5.5.2.2 Mid-Level Graph ADL

```
--
-- Graph sub.adl: Define global variables and store maximum
-- attribute value from current graph
--

int    : sub_node_data;

graph  : node_data;
graph  : DataArcValue, CtlArcValue;

parent : parent_val;
parent : consume0, consume1, produce0;

sub_node_data = node_data + parent_val - 13;

synthesize
   graph_user_integer = max(node_user_integer);
end synthesis
```

Figure 5.9: Mid-Level Graph ADL File

5.5.2.3 Mid-Level Nodes ADL

```
--
-- Node Sub:FixIt1: set user attribute to
-- global variable value plus current attribute
-- value; inherit port attribute from parent node
--

graph : consume0;
graph : sub_node_data;

node_user_integer = node_user_integer + sub_node_data;
token_consume_rate(in0) = consume0;
```

Figure 5.10: First Leaf Node ADL File

```
--
-- Node Sub:FixIt2: set user attribute to
-- global variable value plus current attribute
-- value; inherit port attribute from parent node
--

graph : consume1;
graph : sub_node_data;

node_user_integer = node_user_integer + sub_node_data;
token_consume_rate(in0) = consume1;
```

Figure 5.11: Second Leaf Node ADL File

```
--
-- Node Sub:SubSub: set user attribute to global value;
-- place port attributes in variables for inheritance
--

int : consume0, consume1;
int : produce1;
int : parent_val;

graph : sub_node_data;
graph : produce0;

node_user_integer = node_user_integer + sub_node_data;
token_produce_rate(out0) = produce0;
consume0 = token_consume_rate(in0);
consume1 = token_consume_rate(in1);
produce1 = token_produce_rate(out1);
parent_val = 33;

synthesize
   node_user_integer = node_user_integer + max(node_user_integer);
end synthesis
```

Figure 5.12: Subgraph Internal Node ADL File

5.5.2.4 Mid-Level Arcs ADL

The arcs in the mid-level graph use the same ADL files as the arcs in the top-level graph, as well as the ADL for control arcs defined below.

```
--
-- Arcs of type control: set user attribute to
-- global value plus current attribute value
--

graph : CtlArcValue;

arc_user_integer = arc_user_integer + CtlArcValue;
```

Figure 5.13: Control Arc ADL File

5.5.2.5 ADLEVAL Results

Table 5.15: ADLEVAL Results for Mid-Level Graph

Component Name	*Attribute*	*Initial Value*	*After Inheritance*	*After Synthesis*
Graph file	*graph_user_integer*	—	—	97
Node FixIt1	*node_user_integer*	5	41	41
Node FixIt1, inport 0	*token_consume_rate*	0	2	2
Node FixIt2	*node_user_integer*	6	42	42
Node FixIt2, inport 0	*token_consume_rate*	0	3	3
Node SubSub	*node_user_integer*	7	43	97
Node SubSub, outport 0	*token_produce_rate*	0	4	4
Arc control0	*arc_user_integer*	3	13	13
Arc data0	*arc_user_integer*	4	9	9
Arc data1	*arc_user_integer*	4	9	9
Arc post0	*arc_user_integer*	5	10	10
Arc pre0	*arc_user_integer*	6	11	11
Arc pre1	*arc_user_integer*	6	11	11

5.5.3 Bottom-Level Graph

Refer to Figure 5.14, the *Bottom-Level ADAS Graph,* Figure 5.15, the *Bottom-Level Graph ADL File,* Figure 5.16, the *Third Leaf Node ADL File,* Figure 5.17, the *Send Control Node ADL File,* Figure 5.18, the *Send Data Node ADL File,* and Table 5.16, *ADLEVAL Results for Bottom-Level Graph.*

5.5.3.1 ADAS Graph

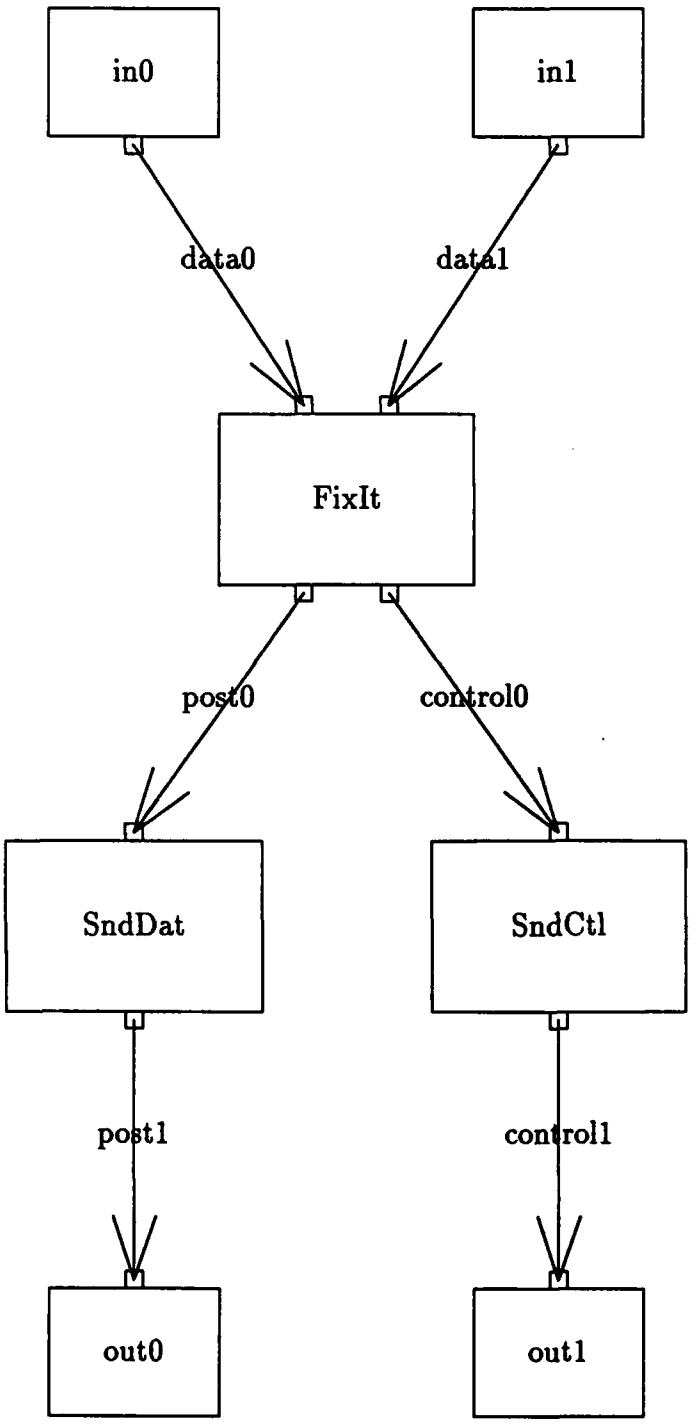

Figure 5.14: Bottom-Level ADAS Graph

5.5.3.2 Bottom-Level Graph ADL

```
--
-- Graph subsub.swg: Define global variables and store maximum
-- attribute value from nodes in current graph
--

int    : subsub_node_data;

graph  : sub_node_data;
graph  : DataArcValue, CtlArcValue;

parent : parent_val;
parent : consume0, consume1, produce1, produce0;

subsub_node_data = sub_node_data + parent_val - 25;

synthesize
   graph_user_integer = max(node_user_integer);
end synthesis
```

Figure 5.15: Bottom-Level Graph ADL File

5.5.3.3 Bottom-Level Nodes ADL

```
--
-- Node Sub:SubSub:FixIt: set user attribute to
-- global variable value plus current attribute
-- value; inherit port attributes from parent node
--

graph : consume0, consume1;
graph : subsub_node_data;

node_user_integer = node_user_integer + subsub_node_data;
token_consume_rate(in0) = consume0;
token_consume_rate(in1) = consume1;
```

Figure 5.16: Third Leaf Node ADL File

```
--
-- Node Sub:SubSub:SndCtl: set user attribute to
-- global variable value plus current attribute
-- value; inherit port attribute from parent node
--

graph : produce1;
graph : subsub_node_data;

node_user_integer = node_user_integer + subsub_node_data;
token_produce_rate(out0) = produce1;
```

Figure 5.17: Send Control Node ADL File

```
--
-- Node Sub:SubSub:SndDat: set user attribute to
-- global variable value plus current attribute
-- value; inherit port attribute from parent node
--

graph : produce0;
graph : subsub_node_data;

node_user_integer = node_user_integer + subsub_node_data;
token_produce_rate(out0) = produce0;
```

Figure 5.18: Send Data Node ADL File

5.5.3.4 Bottom-Level Arcs ADL

The arcs in the bottom-level graph use the same ADL files as the arcs in the top- and mid-level graphs.

5.5.3.5 ADLEVAL Results

Table 5.16: ADLEVAL Results for Bottom-Level Graph

Component Name	*Attribute*	*Initial Value*	*After Inheritance*	*After Synthesis*
Graph file	*graph_user_integer*	—	—	54
Node FixIt	*node_user_integer*	8	52	52
Node FixIt, inport 0	*token_consume_rate*	0	5	5
Node FixIt, inport 1	*token_consume_rate*	0	6	6
Node SndCtl	*node_user_integer*	10	54	54
Node SndCtl, outport 0	*token_produce_rate*	0	7	7
Node SndDat	*node_user_integer*	9	53	53
Node SndDat, outport 0	*token_produce_rate*	0	4	4
Arc control0	*arc_user_integer*	7	17	17
Arc control1	*arc_user_integer*	7	17	17
Arc data0	*arc_user_integer*	8	13	13
Arc data1	*arc_user_integer*	8	13	13
Arc post0	*arc_user_integer*	9	14	14
Arc post1	*arc_user_integer*	9	14	14

6. GRAPH TRANSFORMATION SYSTEM

6.1 Overview

The AIVD Graph Transformation Systems (GTS) transforms a software graph supplied by the user according to transformation rules. The transformation rules are developed by the user in the form of ADAS software graphs.

6.1.1 Execution

GTS is invoked as a stand-alone tool while running Prolog. The command to start Prolog, **quintus_engine**, should also load the compiled GTS code, **gtsrun**. During a Prolog session, the command **testgts_all** specifies the program, the parameter *ingraph* specifies the application graph to be transformed, *outgraph* specifies the name of the result graph, *rulesgraph* specifies the transformation rule base to be used in making the transformation, and *prtlvl* is an integer (0-4) that specifies the level of information to be sent to the standard output.

GTS can be invoked in three levels of interaction: *automatic*, *semi-automatic* and *controlled*. Upon entering Prolog with the compiled **gtsrun**, the user will be prompted as to which level of interaction he prefers to use in executing the program. An example session of GTS follows. The typewriter font represents the system's

prompts and responses, whereas the bold font represents the inputs of the user. The example session is in *controlled* mode with *prtlvl* set to zero.

```
$ quintus_engine dua0:[aivd.src.gts.oct10]gtsrun

Quintus Prolog Release 2.0 (VAX/VMS)
Copyright (C) 1987, Quintus Computer Systems, Inc. All rights reserved.
1310 Villa Street, Mountain View, California (415)965-7700

|?- testgts_all('ingraph','outgraph','rulesgraph',prtlvl).

LOADING INPUT GRAPH FILE: ingraph

**************************************************
Please enter design packages in list form:
|: [].

**************************************************
Please enter hardware types or classes in list form:
|: [].

**************************************************
Print Levels:
0 Minimum Printing
1 Sorted rules; some in-process printing.
2 All rules; additional in-process printing; warnings.
3 Additional transformation details.
4 Additional pattern matching details.

The Print Level is now:
0 :  Minimum Printing

Enter:  #.<CR> to change debugging level;
^C for prolog trace options;
c.<CR> to continue: c.

LOADING TRANSFORM RULES GRAPH

ORDERING TRANSFORM RULES

*** BEGINNING OR CONTINUING GRAPH TRANSFORM PROCESS ***

Target graph for transformation: ingraph
```

```
Enter:  c.<CR> to begin search for pattern match;
o.<CR> to output transformed graph to named file;
h.<CR> to halt and output to normal output file;
a.<CR> to abort.
|: c.

Enter:  #.<CR> to change debugging level;
^C for prolog trace options;
c.<CR> to continue: c.

The following match was found:...
```

6.1.2 Notation

- The user's graph before transformation will be called the *application graph.*
- The user's graph after transformation will be called the *result graph.*
- The left hand side of a transformation rule will be called the *pattern graph.*
- The right hand side of a transformation rule will be called the *transform graph.*

A pattern graph node that has exactly one associated transform graph node will be called *external.* External nodes link the pattern graph and the transform graph to the original application graph. All other nodes in the pattern graph will be called *not external.* Only external nodes of a pattern graph will be allowed to have ports with unattached arcs.

6.2 GTS Inputs

6.2.1 Transformation Rule Base

The transformation rule base structure of taxonomy is shown in Figure 6.1. A transformation rule base file is an ADAS graph file which contains nodes whose subgraphs are pattern graphs and transform graphs. The parent node of a transform graph is associated with the parent node of a pattern graph through an arc connecting the outport of the pattern graph's parent node and the inport of the transform graph's parent node. A pattern graph may have one or more transform graphs associated with it.

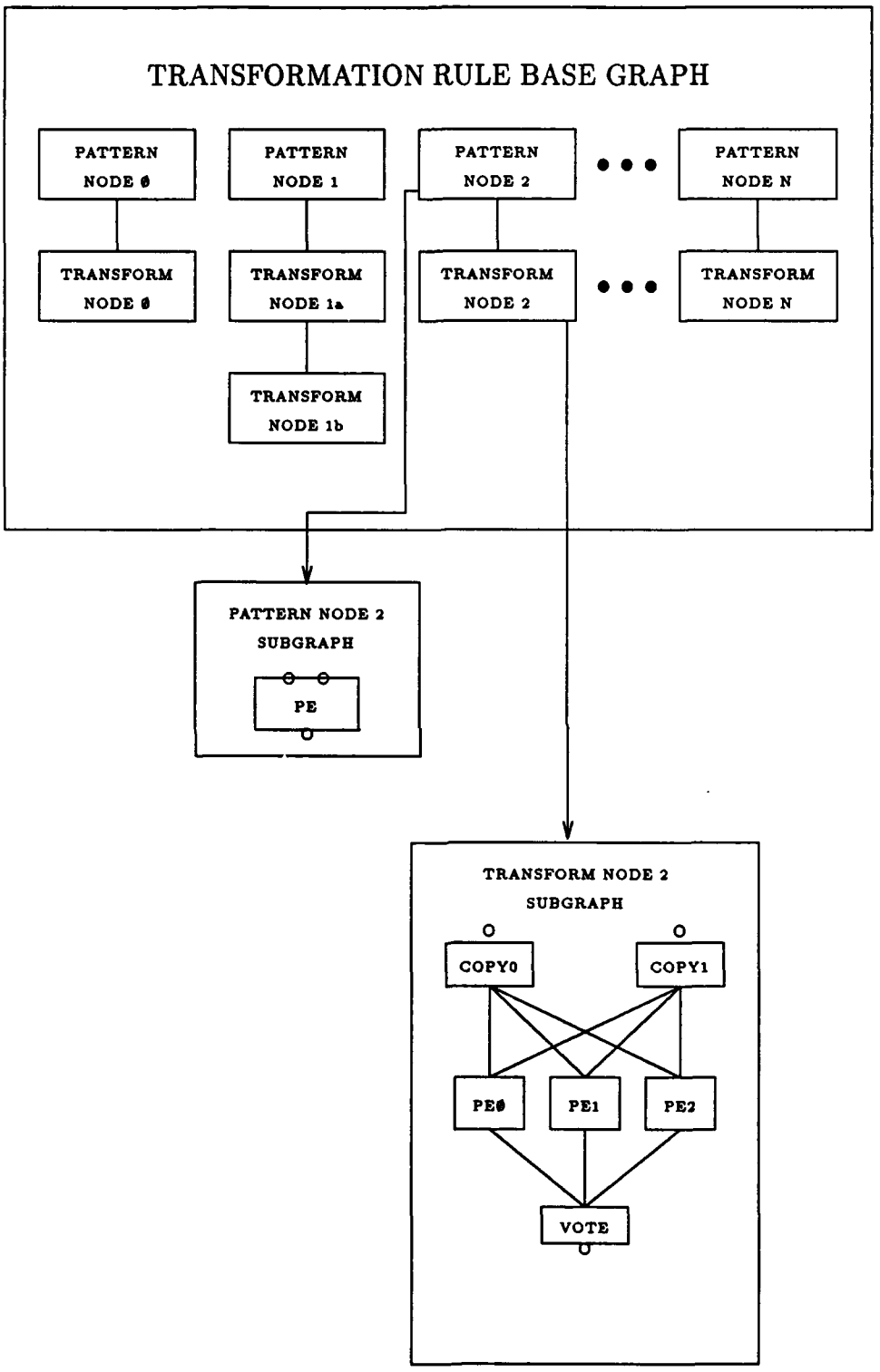

Figure 6.1: Graph Transformation System Taxonomy

6.2.2 Application Graph

The application graph attributes indicated contain the following:

graph_user_file_name	The name of the ADL file associated with the graph.
graph_user_text	The name of the help file associated with the graph.
node_user_file_name	The name of the ADL file associated with the node.
node_user_text	The name of the help file associated with the node.
arc_user_file_name	The name of the ADL file associated with the arc.
arc_user_text	The name of the help file associated with the arc.

6.2.3 Pattern Graph

1. A pattern graph is not allowed to have any graph ports. It may have nodes with ports with no incident arcs.

2. Each node in the pattern graph must be given a unique positive value for its *graph_port_number* attribute, and that attribute must be given a *status* of **N** for *no match required.* This may be accomplished through the **edit status node** *nodename* command.

3. Each inport in the pattern graph must be given a unique identifier for its *inport_id* attribute, and that attribute must be given a *status* of **N** for *no match required.* This may be accomplished through the **edit status node** *nodename* command.

4. Each outport in the pattern graph must be given a unique identifier for its *outport_id* attribute, and that attribute must be given a *status* of **N** for *no match required.* This may be accomplished through the **edit status node** *nodename* command.

5. Each arc in the pattern graph must be given a unique identifier for its *token_units* attribute, and that attribute must be given a *status* of **N** for *no match required.* This may be accomplished through the **edit status arc** *arcname* command.

6. In a pattern graph, the *status* facet of the attribute is used to indicate conditions in matching the pattern graph against the application graph.

   - **M** or *match required* implies that the pattern node (or arc) can only be matched with application nodes (or arcs) that have the same attribute value.
   - **N** for *no match required* implies that the pattern node can be matched with any application node regardless of its attribute value.
   - **P** is not used.

6.2.4 Transform Graph

1. Transform node attribute *graph_port_number* is used to indicate that a particular transform node is *associated* with a particular pattern node.
   - A value 0 means *new node* or *no association.*
   - A value of n means that the transform graph node is associated with the pattern graph node that has n as its *graph_port_number* value.
2. Transform inport attribute *inport_id* is used to indicate that a particular transform inport is *associated* with a particular pattern inport.
   - A value N means *new inport* or *no association.*
   - A value of In means that the transform graph node inport is associated with the pattern graph node inport that has In as its *inport_id* value.
3. Transform outport attribute *outport_id* is used to indicate that a particular transform outport is *associated* with a particular pattern outport.
   - A value N means *new outport* or *no association.*
   - A value of On means that the transform graph node outport is associated with the pattern graph node outport that has On as its *outport_id* value.
4. Transform arc attribute *token_units* is used to indicate that a particular transform arc is *associated* with a particular pattern arc.
   - A null value means *new arc* or *no association.*
   - A value of ID means that the transform graph arc is associated with the pattern graph arc that has ID as its *token_units* value.
5. For each transform attribute, the *modifiable* facet of the attribute is used to indicate whether, in the transformed node (or arc), the value of the attribute is to be copied from the matched application node (or arc) or from the transform graph node (or arc).
   - **N** or *not modifiable* implies copy from the matched application node or arc.
   - **M** or *user modifiable* implies copy from the associated transform node or arc.
   - **P** or *program modifiable* is not used.

6.3 GTS Processing

6.3.1 The Matching Process

A pattern graph matches an application graph if the following hold:

1. Each node N_p in the pattern graph matches a corresponding node N in the application graph; i.e.,

   (a) For each node attribute $a(N\_p)$ of N_p,

   if $status(N\_p,\ a) = M$
   then $a(N\_p) = a(N)$

   (b) For each input port attribute $a(N\_p,\ k)$ of N_p,

   if $status(N\_p,\ k,\ a) = M$
   then $a(N\_p,\ k) = a(N,\ k)$

   (c) For each output port attribute $a(N\_p,\ k)$ of N_p,

   if $status(N\_p,\ k,\ a) = M$
   then $a(N\_p,\ k) = a(N,\ k)$

   Where status $(N\_p,\ a)$ means the status of attribute a of node N_p and status $(N\_p,\ k,\ a)$ means the status of attribute a of the k^{th} inport or outport, as applicable.

2. Each arc A_p in the pattern graph matches a corresponding arc A in the application graph; i.e.,

   (a) The arc sources must match, that is,

   if $source(A\_p) = N\_p$ and $source(A) = N$
   then N_p must match N

   (b) The arc sinks must match, that is,

   if $sink(A\_p) = N\_p$ and $sink(A) = N$
   then N_p must match N

   (c) The arc attributes must match, that is,

   if $status(A\_p,\ a) = M$
   then $a(A\_p) = a(A)$

3. All arcs in the application graph that are incident to a node that matches the pattern graph will either connect two nodes that match pattern nodes, or will connect a node that does not match a pattern node to a node that matches

an external node. This rule guarantees that all arcs in the application graph that connect the matched portion of the graph to the rest of the graph will also occur in the result graph.

6.3.2 The Replacement Process

1. The equivalent of block move operations are performed to create enough space around the matched section of the application graph to accommodate the transform graph.
2. All nodes in the application graph that match nodes in the pattern are deleted.
3. For each node in the transform graph a node is added to the application graph. If the node has no association, the transform graph node is copied into the appropriate location in the application graph.
4. For each node in the transform graph that has an associated node, the attributes of the application graph node and the transform graph node is merged using the attribute status as a guide.
5. Arcs in the application graph which match arcs in the pattern graph are deleted.
6. For each arc in the transform graph, an arc is added to the application graph.
7. If an arc in the transform graph has no association, the transform graph arc attributes are copied. For each arc in the transform graph that has an associated arc, the attributes of the application graph arc and the transform graph arc are merged using the attribute status as a guide.
8. Each arc in the application graph which was connected to a node not matched by the pattern graph at one end of the arc and was connected to a node matched by some pattern graph node at the other end of the arc, is reconnected to a new node, using the node inport or outport association as a guide.

6.4 GTS Outputs

The output of GTS is a transformed ADAS graph, with attributes modified as described in the preceding section on GTS Processing.

6.5 An Example

Figure 6.2 shows the example ADAS application graph. Based on the topology of the application graph, there are three possible matches for the pattern graph, one for each row of the graph. However, only one of the matches satisfies all the conditions required for a match.

Figure 6.3 shows the example pattern graph, which is a simple three-stage pipeline graph. Note the extra input and output ports on the external nodes of the pattern, namely the first and last stages of the pipeline. The following attributes are being used for matching nodes and input and output ports:

node_user_file_name
in_token_data_type
out_token_data_type

The following attributes are being used for matching arcs:

arc_color
arc_user_file_name

Node `A` in the pattern graph could match `A1`, `A2`, or `A3` in the application graph, and similarly `C` could match `C1`, `C2`, or `C3`. However, node `B` could match `B1` or `B2`, but not `B3`, since `B` is not an external node and `B3` has an arc incident that does not match an arc in the pattern. Furthermore, arc `arrow3` in the pattern graph cannot match arc `arrow30` in the application graph since their colors do not match. This forces exactly one match, with `A` matching `A2`, `B` matching `B2`, and `C` matching `C2`.

Figure 6.4 shows the corresponding transform graph. Notice that a new node is created, and so is a new arc. The transform files associate transform node attributes with pattern node matches as follows:

`A` gets attributes from the node matched with `A`
`B0` gets attributes from the node matched with `B`
`B1` gets attributes from the node matched with `C`
`C` gets attributes from the node matched with `C`

Attributes are shared for input ports as follows:

`A(0)` gets attributes from the input port matched with `A(0)`
`A(1)` gets attributes from the input port matched with `A(1)`
`B0(0)` gets attributes from the input port matched with `B(0)`
`B1(0)` gets attributes from the input port matched with `C(0)`
`C(0)` gets attributes from the input port matched with `C(0)`
`C(1)` gets attributes from the input port matched with `C(1)`

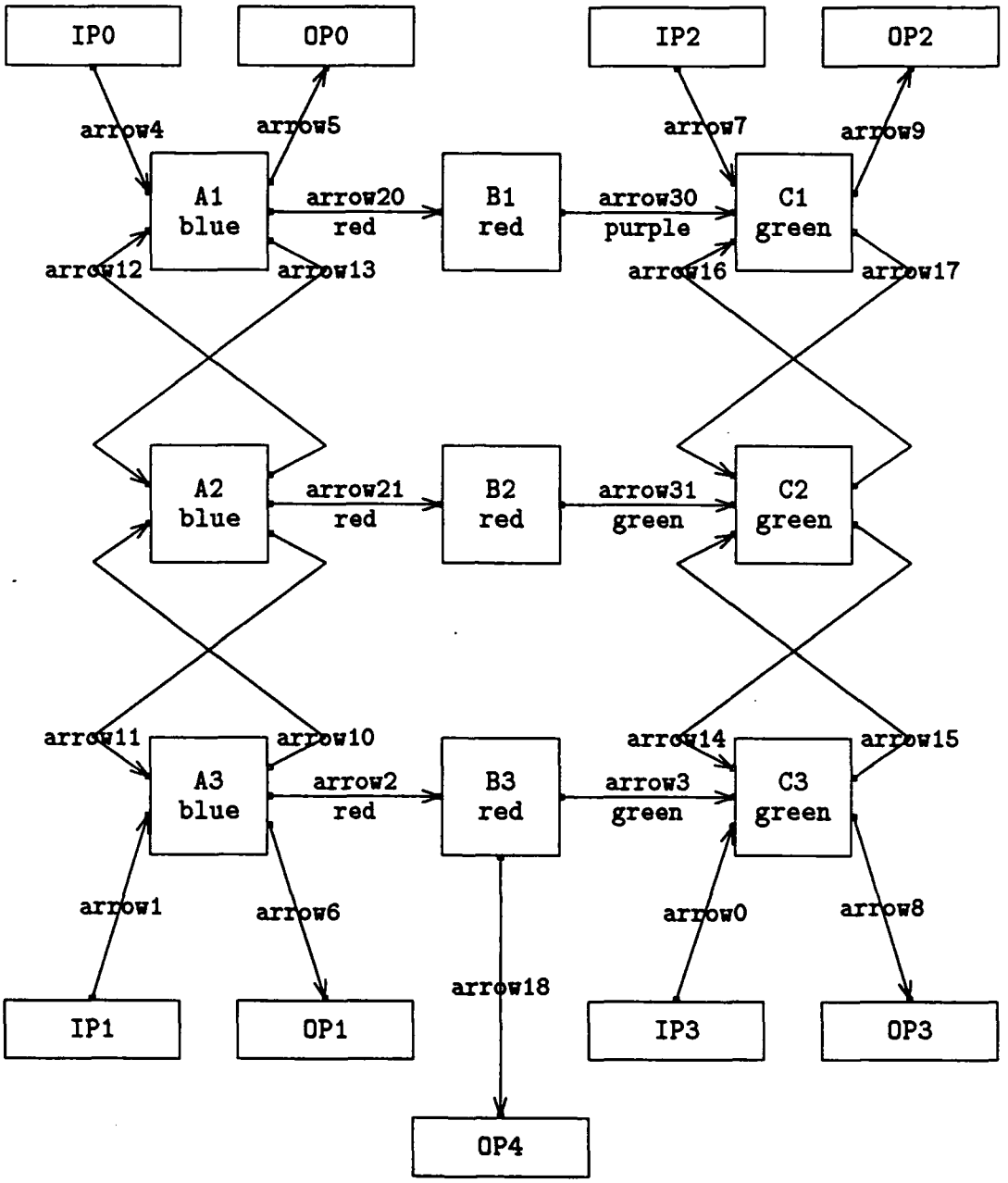

Figure 6.2: The Example Application Graph

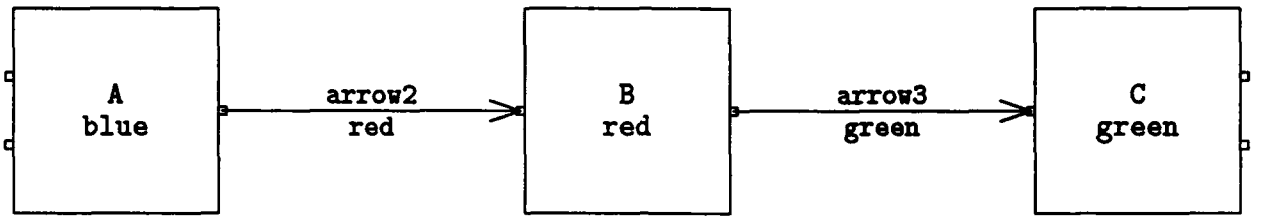

Figure 6.3: The Example Pattern Graph

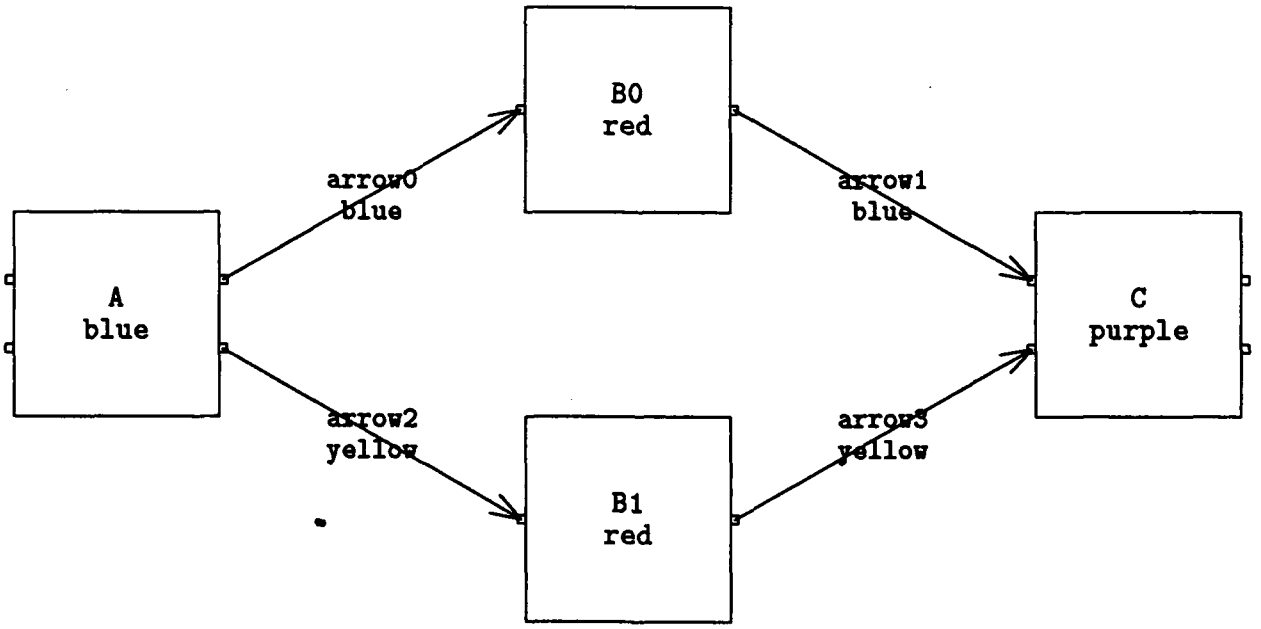

Figure 6.4: The Example Transform Graph

Attributes are shared for output ports as follows:

A(0) gets attributes from the output port matched with A(1)
A(1) gets attributes from the output port matched with A(1)
B0(0) gets attributes from the output port matched with B(0)
B1(0) gets attributes from the output port matched with C(0)
C(0) gets attributes from the output port matched with C(0)
C(1) gets attributes from the output port matched with C(1)

The inheritance of *node_color* attributes is set as follows:

A gets its color from the node matching pattern node A
B0 gets its color from the transform graph
B1 gets its color from the node matching pattern node C
C gets its color from the transform graph

The inheritance of *arc_color* attributes is set as follows:

arrow0 gets its color from the arc matching pattern arc arrow2
arrow1 gets its color from the transform graph
arrow2 gets its color from the arc matching pattern arc arrow3
arrow3 gets its color from the transform graph

Figure 6.5 shows the result of transforming the original application graph. One match has taken place, and in fact only one match can take place.

6.6 GTS Control Rules

Additional "intelligent" control over the graph transformation process is provided through the use of GTS control rules. GTS control rules may be used for

1. Rule selection — deciding which GTS pattern/transform combinations should be tried.
2. Pattern Matching — providing additional logical capabilities for accepting or rejecting pattern matches.
3. Transform Evaluation — deciding which of a set of transform graphs, all associated with some particular pattern graph, should be used in a given graph transformation.
4. Attribute Calculations — calculating attribute values as functions of prior graph attributes.

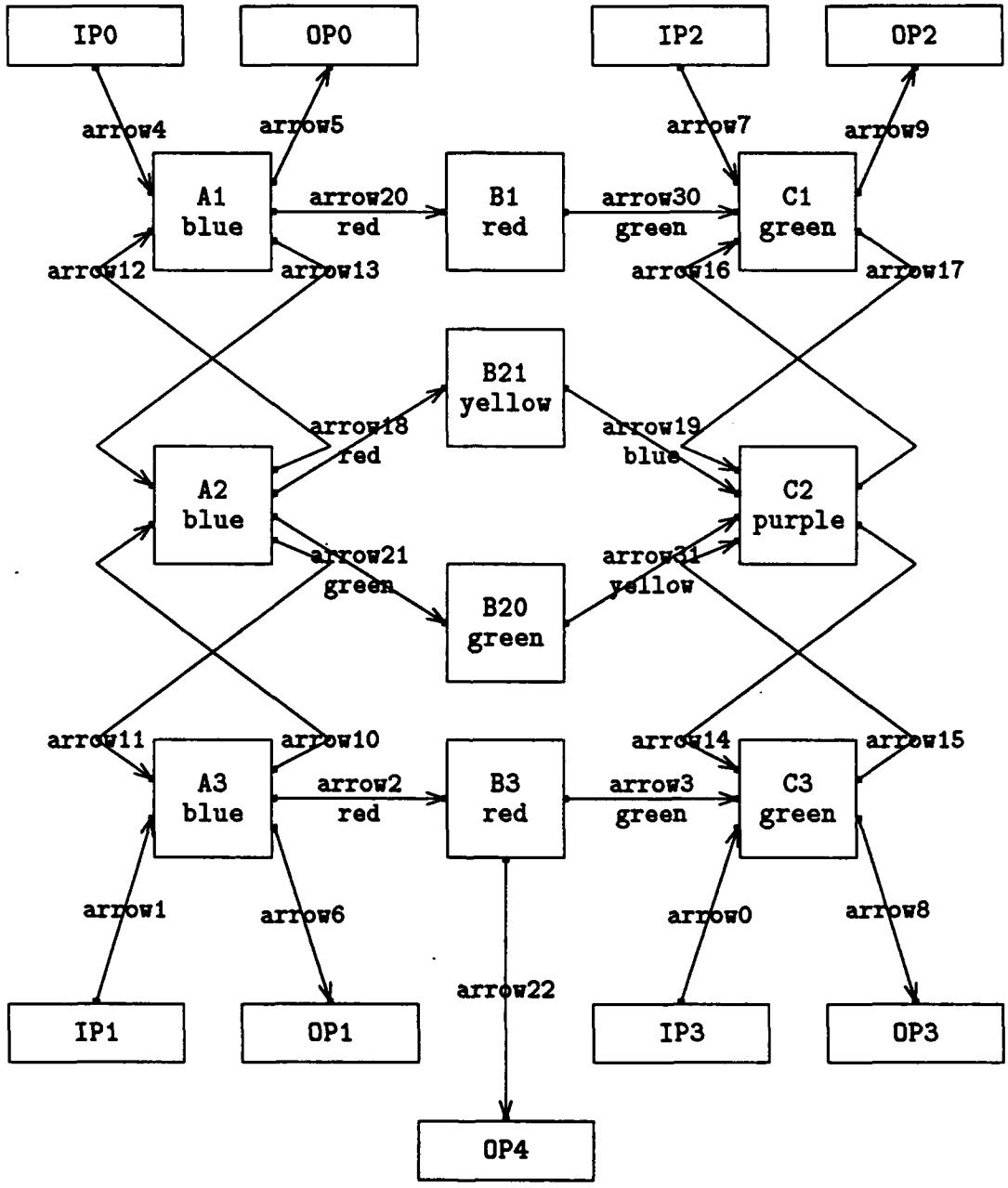

Figure 6.5: The Example Result Graph

6.6.1 Rule Selection

Control rules for rule selection are "built-in" and use rule "class" specifications.. Every rule may have an associated design package class and hardware class.

> Design package classes are specified using the *package_file_name* attribute of the rule graph pattern node. Any number of package names may be specified, separated by a semicolon — no spaces are allowed. For example, the list of names "filter_design" and "4kfilter" are specified using the *package_file_name* attribute value "filter_design;4kfilter."
>
> Hardware classes are specified using the *hw_module* attribute of the rule graph transform node or nodes. Only a single name may be specified for any particular transform. Hardware classes may be either a specific design of hardware or a class of hardware designs for which particular transformations may be appropriate.

In using GTS, a user is asked to input desired design packages and hardware classes. These are input in the form

[<*class_name*>,<*class_name*>,...].

GTS uses these class names by accepting situations of one or more matching name values as "allowing" a controlled situation whenever non-empty values or sets of values are specified. Two situations are controlled in this way: the initial selection of rules (patterns) to be applied and the selection of transforms to be evaluated for use after a pattern match is found. If a rule class specification is empty, or a user does not input any class specification list, the associated class tests are omitted.

GTS also uses rule priority to order all selected rules before trying pattern matches. Rule priority is specified by using the rule graph pattern node attribute *node_user_integer.* Higher values have higher priority.

6.6.2 Pattern Matching

When GTS has succeeded in making a pattern match of a pattern graph to an application graph, as specified in Section 6.3, GTS will then attempt to "accept" the match through use of an associated "match" rule. A match rule may be defined in a file whose filename is specified by the rule graph pattern node attribute *node_user_file_name.* If no match rule is specified, the pattern match is accepted.

Currently, match rules are written in Prolog, in the form

matchrule :- *< rule >*.

A match rule must set a "global" flag "acceptmatch(true)" if a match is to be accepted. The following example, taken from Example 2, illustrates a match rule:

```
/*
--iosfa_m.adl
--Matching rule for iterated overlap-save filter (A).
--Accept a match if the application node matched has an input
--dimension greater than 4096.
       */

matchrule :-
     retractall(acceptmatch(_)),
     (matchnval('F0',inport('I1',cons),V),
       V > 4096,
       assert (acceptmatch(true))
       ;
       assert (acceptmatch(false)) ),
       !.
matchrule.
```

The term "matchnval ('F0',inport('I1',cons),V)" gets the value of the consume attribute of the inport named I1 of the application graph node which matches pattern node F0. In general, the term

matchnval(*<pattern_node>*, *<attribute>*, **Value**)

gets the value of *<attribute>* of the application graph node which matches *<pattern_node>*. *<attribute>* may be either a simple attribute name, or may be

inport (*<inport_name>*, *<inport_attribute>*)
or outport (*<outport_name>*, *<outport_attribute>*)

The Prolog construct

(*<terms1>*;*<terms2>*)

is an "or" construct; if *<terms1>* is not "proven," *<terms2>* will be tested. Thus, in the example, if the value of cons is greater than 4096, the match will be accepted; if the value of cons is less than or equal to 4096, the match will not be accepted.

6.6.3 Transform Evaluation

After a pattern match has been accepted, GTS will evaluate whether to make a transform. If the rule graph pattern node has only one transform node associated with it and no evaluation rule is specified, the transform will be executed. If an evaluation rule is specified, the transform may or may not be executed, depending on the evaluation rule. If more than one transform node is specified for the pattern node, the transform for which the associated evaluation rule defines the highest "evaluatevalue" value is executed.

An evaluation rule may be specified for a transform graph by defining it in a file which is specified by the rule graph transform node attribute *node_user_file_name*. Evaluation rules are defined in Prolog, in the form

```
evaluaterule :-
        < rule >.
```

An evaluation rule must set a global value "evaluatevalue(<value>)"; the transform for which the value is highest is selected for execution. If only a single transform node is associated with a rule graph pattern node, the transform will be executed if *<value>* is not zero. For example, suppose a particular transform is intended for a pipeline (fft.mult.fft) transformation and is to be accomplished if the datarate lies between limits MinRate and MaxRate. An example of an evaluation rule implementing this logic is

```
/*
--fltl.adl

--Evaluation rule for pipeline (fft.mult.fft)
--filter transformation.

--FrameRate is vectors per second; use node_user_float
--attribute

--MaxRate is maximum datarate, bytes per second,
--MinRate is minimum datarate (use a simpler implementation
--if datarate is less than MinRate).

--Do transform if datarate <= MaxRate and > MinRate.
    */

evaluaterule :-
```

```
retractall(evaluatevalue(_)),
MaxRate = 6500000,
MinRate = 4000000,
matchnval ('F0',usfp,FrameRate),
matchnval ('F0',inport('I1',cons),N_data),
DataRate is N_data * FrameRate,
(   DataRate =< MaxRate,
    DataRate > MinRate,
    assert(evaluatevalue(10)) )
   ;
    assert(evaluatevalue(0)) ).
```

6.6.4 Attribute Calculations

The graph transformation is initially processed as described in Section 6.3.2. After making the transformation, GTS checks to see if an attribute rule is specified. An attribute rule is specified in the same file as used for evaluation rules; that is, the file is specified by the rule graph transform node attribute *node_user_file_name*. If an attribute rule is specified, it is executed to set attribute values which cannot be set using the "merge" operations of Section 6.6.3. An attribute rule is defined in Prolog in the form

```
attributesrule :-
        < rule >.
```

For example,

```
/*
--iosfa_t.adl
--Calculated attributes for iterated overlap-save filter (A)
--transformation.
     */

attributesrule :-
   matchnval('F0',inport('I1',cons),N_data),
   matchnval('F0',inport('I0',cons),N_coef),
   Overlap is N_coef - 1,
   Rem_seg is N_data - 4096 + Overlap,
   Difference is Rem_seg - Overlap,
```

```
setnval('copy0',outport('000',prod),N_data),
setnval('copy0',outport('001',prod),N_data),
setnval('sfn0',inport('II0',cons),N_data),
setnval('fn',inport('II0',cons),N_data),
setnval('sfn0',outport('000',prod),Rem_seg),
setnval('copy1',outport('000',prod),N_coef),
setnval('copy1',outport('001',prod),N_coef),
setnval('filt1',inport('II0',cons),N_coef),
setnval('filt0',inport('II0',cons),N_coef),
setnval('filt0',inport('III',cons),Rem_seg),
setnval('filt0',outport('000',prod),Rem_seg),
setnval('sfn1',inport('II0',cons),Rem_seg),
setnval('sfn1',outport('II0',prod),Difference),
setnval('cct',inport('II0',cons),Difference).
```

The terms "setnval (..." are somewhat the inverse of matchnval(...

setnval(<*transform_node*>,<*attribute*>,Value)

sets the value of <*attribute*> of the new application graph node associated with the <*transform_node*>.

7. THE ALLOCATOR OF SOFTWARE TO HARDWARE

7.1 Overview

The AIVD Allocator of Software to Hardware (ASH) is an enhanced version of the ADAS Version 2.5 mapping tool (also called ASH). ASH reads a software graph data base and attempts to map its nodes onto a hardware graph data base using rules from a mapping rules base. ASH is executed by entering the command

ash *swdbase hdwbase* **-s** *script* **-d** *gterm*

where:

- *swdbase* is the name of the software graph data base,
- *hwdbase* is the name of the hardware graph data base,
- *script* is the name of a script file, and
- *gterm* is the name of the graphics display device.

ASH generates an ADAS script file which sets the *hw_module* attributes of software nodes. Only those nodes with an attribute modification status other than **N** for the *hw_module* attribute are considered for assignment.

Figure 7.1 shows the inputs and outputs of ASH. The user interface for ASH is based on the standard ADAS model, where the current graph of the hierarchy is displayed in a window, and a menu is attached either to the left side or the right side of the graph window. The primary format for ASH outputs is the script file, which can be read by other ADAS tools.

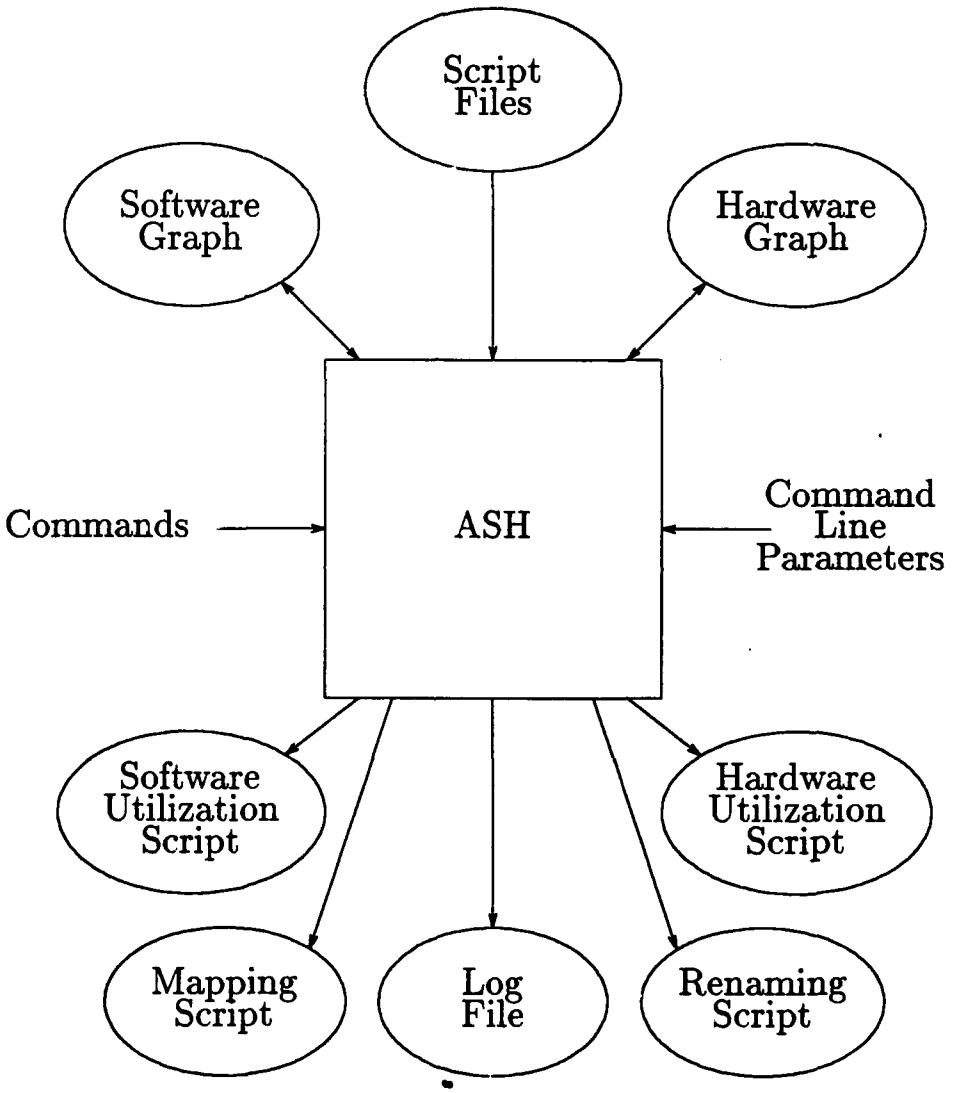

Figure 7.1: Allocator of Software to Hardware Structure

7.2 Input

7.2.1 Parameters

The following parameters are entered as part of the ASH command line.

software graph data base This is the file name for the root graph of the software graph hierarchy.

hardware graph data base This is the file name for the root graph of the hardware graph hierarchy.

script file This is the file name for a script file. The script file can contain specifications of the ASH mapping parameters described in the control command, initial software graph utilization information as generated by GIPSIM for an unconstrained graph, as well as the ADL initialization of graph attributes.

graphics display device This is the identifier for graphics device for the display and the mouse. If ASH is being run from an ASCII terminal, as opposed to a graphics workstation, then the device name is null. If either a workstation or display terminal is being used, then the standard ADAS identifier for the device should be used.

7.2.2 Files

7.2.2.1 ADAS Software Graph

The software graph hierarchy for ASH specifies the software which is mapped to the hardware. ASH reads the data base and flattens the graph before starting the mapping process. ASH uses the following information in the software graph:

1. The connectivity of the software graph is used to constrain the possible mappings. In particular, there must be an arc B in the hardware graph which corresponds to each arc A in the software graph such that $hw\_mod(Source(A)) = Source(B)$ and $hw\_mod(Sink(A)) = Sink(B)$.

2. The attributes that ASH uses for its calculations which are embedded in the software graph data base are:

   - The node hardware module attribute, denoted by $hw\_mod(N)$. Generally, this attribute will be blank when ASH starts working. If it is not blank and the modification status on the attribute is "N", then ASH will treat this

mapping information as locked by the user, and therefore will not modify this mapping.

- The node hardware module_class attribute, denoted by $module\_class(N)$. This attribute is used to determine what set of possible hardware modules would allow a mapping; i.e., if $module\_class(M) = module\_class(N)$, then software node N can be mapped to hardware module M.
- The node utilization attribute, denoted by $U_f(N)$. This attribute should be the result of a GIPSIM simulation of the unconstrained software graph, i.e., a version of the software graph where each node has its own hardware module. This attribute may be stored in the attributes when the graph is loaded, or it may be loaded through a script file. The latter is necessary if the software graph hierarchy has shared subgraphs.

7.2.2.2 ADAS Hardware Graph

The hardware graph hierarchy for ASH specifies the resources to which the software is mapped. ASH reads the data base and flattens the graph before starting the mapping process. ASH uses the following information in the hardware graph:

1. The connectivity of the hardware graph is used to constrain the possible mappings. In particular, there must be an arc path B in the hardware graph which corresponds to each arc A in the software graph such that $hw\_mod(Source(A)) = Source(B)$ and $hw\_mod(Sink(A)) = Sink(B)$.
2. The node hardware module class attribute is used to determine what set of possible hardware modules would allow a mapping; i.e., if $module\_class(M) = module\_class(N)$, then software node N can be mapped to hardware module M.

7.2.3 Commands

The behavior of ASH can be controlled interactively through the command set listed below. The functions of the commands are described in the Command Processing section.

control	This command determines the values of the parameters for the mapping process.
edit	This is the usual ADAS command for editing graph hierarchies.
environ	This is the usual ADAS command for setting the display environment for the ADAS graphs.
hardware	This command allows the user to switch to browse through the hardware graph hierarchy.
log	This command turns on or off the log of moves made during the mapping process.
macro	This is the usual ADAS command for creating and deleting macro commands.
map	This command performs the mapping of software to hardware based on the control options defined by the user.
output	This command generates a script file defining the current mapping and the expected utilizations of software and hardware.
quit	This is the usual ADAS command for moving back up the software (or hardware) graph hierarchy or out of ASH.
save	This is the usual ADAS command for saving a graph hierarchy.
script	This is the usual ADAS command for reading a script file.
software	This command allows the user to browse through the software graph hierarchy.
stats	This is the usual ADAS command for displaying attributes of the software or hardware graph.
subgraph	This is the usual ADAS command for moving down the software or hardware graph hierarchy.
window	This is the usual ADAS command for changing the window.

7.3 Processing

7.3.1 Performance Estimators

The estimated utilization of a free software node N is denoted by $\hat{U}_f(N)$. This utilization is obtained by executing GIPSIM on a version of the software graph where each node is provided with its own hardware resource.

The estimated free utilization of a hardware module M is denoted by $\hat{U}_f(M)$. It is calculated by the formula

$$\hat{U}_f(M) = \sum_{hdw\_mod(n)=M} \hat{U}_f(n)$$

The maximum free utilization of a hardware module is denoted by $\hat{U}_{max}$. It is calculated by the formula

$$\hat{U}_{max} = \max_{M \in \text{hwg}} (\hat{U}_f(M))$$

The estimated utilization of a constrained software node N is denoted by $\hat{U}_c(N)$. If $\hat{U}_{max} \leq 1$, then

$$\hat{U}_c(N) = \hat{U}_f(N)$$

Otherwise, if $\hat{U}_{max} > 1$, then

$$\hat{U}_c(N) = \hat{U}_f(N)/\hat{U}_{max}$$

This is the utilization that is written into the software utilization script file and is displayed by the **stats** command.

The estimated constrained utilization of a hardware module M is denoted by $\hat{U}_c(M)$. It is calculated by the formula

$$\hat{U}_c(M) = \sum_{hdw\_mod(n)=M} \hat{U}_c(n)$$

This is the utilization that is written into the hardware utilization script file and is displayed by the **stats** command.

7.3.2 The ASH Mapping Algorithm

Figure 7.2 shows the ASH mapping algorithm. This is the algorithm executed when the data bases have been loaded. script files have been run to set the initial attributes, and the user has defined the control options for the mapping algorithm. After the initial random mapping of the software to the hardware, the algorithm goes into a series of iterations of the two major functions:

- enforcing the connectivity constraint, i.e., making sure that each software arc A has a unique corresponding hardware arc B such that $hw\_mod(Source(A)) = Source(B)$ and $hw\_mod(Sink(A)) = Sink(B)$.
- minimizing the maximum hardware module utilization, which in turn minimizes the maximum software node utilization.

The number of iterations of this major loop is controlled by the *iterations* parameter set with the **control** command.

Each of the two major functions described above goes through its own iteration process of prioritizing software nodes to be moved, selecting hardware modules to serve as the destination of the software node move, and then making the map and updating the internal data structures. The numbers of iterations of these processes is also controlled by the user through the *enforce* and *move percentage* parameters set with the **control** command.

7.3.2.1 Creating an Initial Map

As soon as the user has entered the **map** command, ASH will perform an initial random mapping to start the optimization process. No attempt is made by the initialization routines to enforce connectivity in the mapping of software to hardware, but the initial mapping does meet the *module_class* constraints, .i.e., if software node N is mapped to hardware node M, then $module\_class(N) = module\_class(M)$.

7.3.2.2 The Major Iteration

ASH proceeds to optimize the mapping by changing the mappings of software nodes. This change is made in a series of iterations. Each iteration first tries to enforce the connectivity constraints and then attempts to reduce the maximum utilization of the hardware modules. One of the control parameters that the user can define is the number of iterations to be performed before the optimization process is halted.

7.3.2.3 Enforce Connectivity

During the process of enforcing connectivity, ASH attempts to minimize the number of software arcs that violate the connectivity constraint.

The first step in this process is to determine the priority order in which software nodes will be moved. The order is determined by the number of connectivity violations on arcs that are incident to the nodes. Thus a node N_1 which has more connectivity violations on its incident arcs than a node N_2 will have a higher priority order than

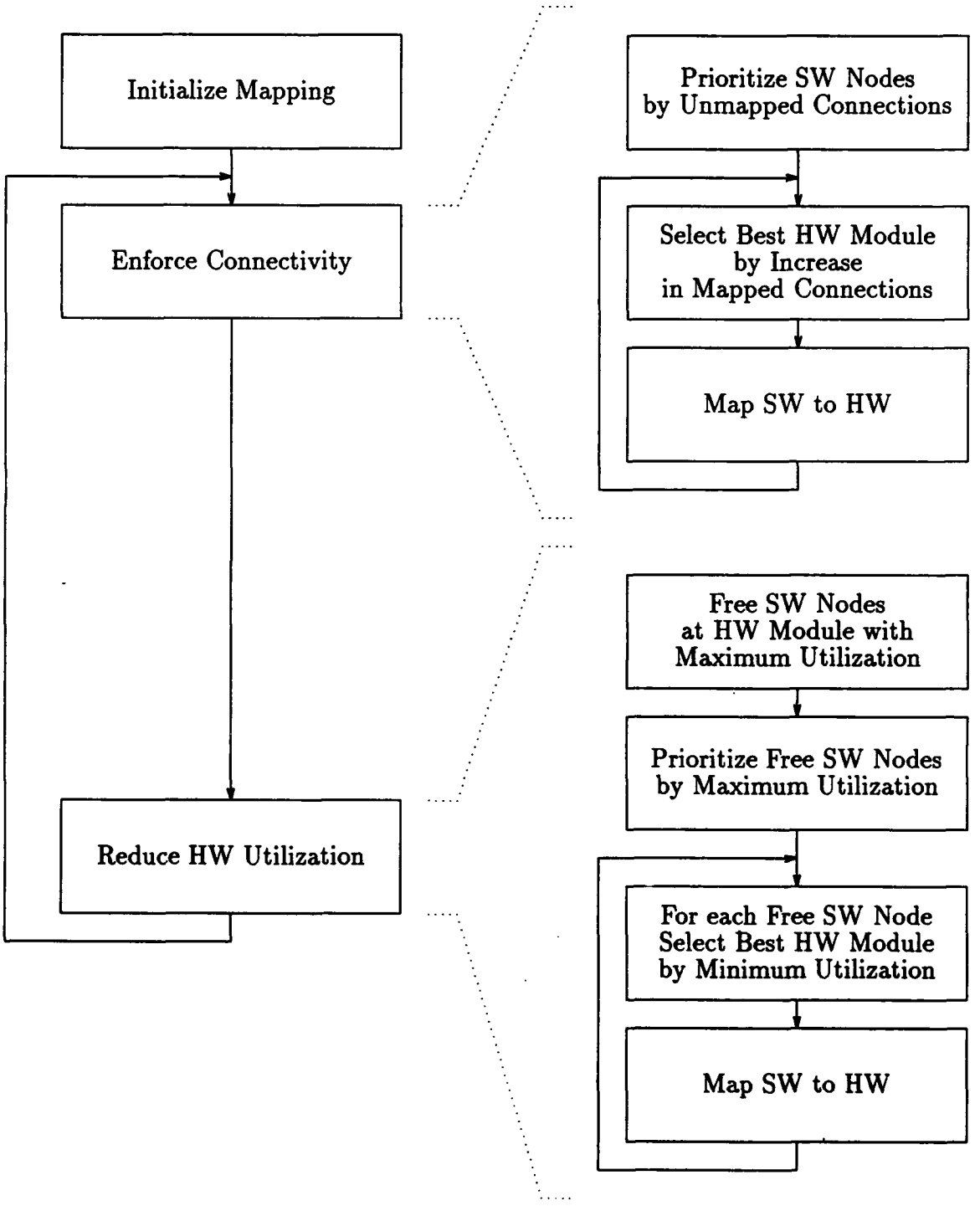

Figure 7.2: ASH Mapping Algorithm

N_2. The ordering also takes into account the amount of time since the node was last moved. The larger this time interval, the higher the priority assigned to moving the node.

Once the software nodes have been assigned priority orders, each node N in order is moved by selecting the hardware module M which will cause the greatest decrease in the number of unmapped software edges when N is moved to M.

Finally, the highest priority software node is mapped to the selected hardware module and is removed from the hardware module it previously was mapped to as well as from the priority list. The next highest priority software node is then picked and a hardware module is again selected.

The process stops when connectivity is achieved (in which case control passes to the second part of the iteration), or a fixed number of connectivity iterations is exceeded. If the priority list is emptied before then, the list is reloaded by recalculating the priorities of all the software nodes based on the mapping which resulted from the last connectivity move. The number of connectivity iterations is a parameter which can be selected by the user through the **control** command.

7.3.2.4 Reduce Maximum Hardware Utilization

The second part of the mapping iteration is reducing hardware utilization. The first step of this part is to priority order all the software nodes based upon the utilization of the hardware modules which they occupy. Software nodes in heavily utilized hardware modules receive a high priority for being moved.

Next, a hardware module is selected which will have a utilization lower than the current maximum hardware utilization once the software node with the highest priority is mapped to it. This movement will reduce the utilization of the most heavily utilized hardware module but will not increase the utilization of the selected hardware module to the same level, thus removing potential bottlenecks.

The highest priority software node is mapped to the selected hardware module and is removed from the hardware module it previously was mapped to as well as from the priority list. This usually causes some software edges to become unmapped and connectivity can no longer be guaranteed. When connectivity is again being re-established in Enforce Connectivity process, this software node is given a low priority for being moved in order to prevent it from being moved back to its original position as an easy way to restore connectivity (that explains the reason why newly moved nodes receive low priority for movement). Upon the completion of this step, control returns to moving the new high priority software node to its best destination hardware module. The Reduce Utilization process ends when a fixed percentage of the original priority list has been moved (say 30%).

7.3.3 Command Processing

The behavior of ASH can be controlled interactively through a command set, which is patterned on the GIPSIM command set. The following sections describe this set of commands.

7.3.3.1 The Control Command

This command determines the values of the parameters for the mapping process. There are three parameters:

iterations	the number of major loop iterations to be performed by the mapping algorithm,
move percentage	the percentage of nodes to be moved from the node with the greatest utilization, and
enforce	number of iterations of the enforce connectivity module.

7.3.3.2 The Edit Command

This is the usual ADAS command for editing graph hierarchies. The command **edit node** allows the user to edit all the attributes for a node and its input and output ports. The command **edit arc** allows the user to edit all the attributes for an arc. The command **edit select** allows the user to edit the values for a particular attribute either for all nodes, all arcs, or selected nodes, or selected arcs. The command **edit global** allows the user to broadcast a value of a particular attribute to all nodes or arcs which share the same template.

7.3.3.3 The Environment Command

This is the usual ADAS command for setting the display environment for the ADAS graphs. The command **environ display** allows the user to either turn on or off the display of the graph. The command **environ grid** allows the user to either turn on or off the display of the grid on which the nodes and the arc joints are positioned. The command **environ node_labels** allows the user to either turn on or off the display of the labels on the nodes of the graph. The command **environ arc_labels** allows the user to either turn on or off the display of the labels on the arcs of the graph. The command **environ menu** allows the user to position the menu either on the left or the right of the graph window.

7.3.3.4 The Hardware Command

This command changes the context from the software graph hierarchy to the hardware graph hierarchy. If the user is in an ADAS software graph, then this command first gives the user the option of saving the software graph (either the **current** graph or the **hierarchy** below the current graph) and then changes the context to the root hardware graph. If the user is already in the hardware graph context, then an error message is printed and the command does nothing.

7.3.3.5 The Log Command

This command turns on or off the log of moves made during the mapping process. The log indicates whether the mapping was done during the **enforce connectivity** process or during the **minimize utilization** process. Each log record describes the move of a software node to a hardware module. The format for a log is shown in Figure 7.6.

7.3.3.6 The Macro Command

This is the usual ADAS command for creating and deleting macro commands. The command **macro add** *macro name* `ADAS command` allows the user to add a new command called *macro name* to the top level ASH menu, where the command is defined by an ADAS command string. The macro names must be distinct. The command **macro delete** *macro name* deletes a macro from the top level ASH menu. The command **macro list** lists the current macro definitions.

7.3.3.7 The Map Command

This command creates a mapping as described in the ASH Mapping Algorithm section. Two messages are produced: the first indicating that the initial random mapping has been completed, and the second indicating that all of the major iterations have been completed.

7.3.3.8 The Output Command

This command enables the user to generate one or more cutputs resulting from the excuted mapping.

mapping Produces a script file setting the *hw_module* attribute of the nodes in a software graph hierarchy.

hw_names	Produces a script file uniquely naming all of the *node_name* attributes of the hardware components in the hardware graph.
utilization	Produces two script files loading estimated *node_utilization* values for both hardware and software graphs.
report	Produces a report of the assignments made during the mapping session.

7.3.3.9 The Quit Command

This is the usual ADAS command for moving back up the software (or hardware) graph hierarchy. At the root of either the software or hardware graph hierarchy, this command causes an exit from ASH. Otherwise, the command causes ASH to exit from the current graph and return to the parent graph. As in GIPSIM, the command allows the user to save either the current graph or the entire graph hierarchy below and including the current graph.

7.3.3.10 The Save Command

This is the usual ADAS command for saving a graph hierarchy. As in GIPSIM, the command allows the user to save either the current graph or the entire graph hierarchy below and including the current graph.

7.3.3.11 The Script Command

This is the usual ADAS command for reading a script file. The user must specify the name of the script file. Any errors found by ASH during the execution of the script file will cause the execution of the entire script file to be aborted.

7.3.3.12 The Software Command

This command changes the context from the hardware graph hierarchy to the software graph hierarchy. If the user is in an ADAS hardware graph, then this command first gives the user the option of saving the software graph (either the **current** graph or the **hierarchy** below the current graph) and then changes the context to the root software graph. If the user is already in the software graph context, then an error message is printed and the command does nothing.

7.3.3.13 The Statistics Command

This is the usual ADAS command for displaying attributes of the graph. When utilization is displayed, the utilization values are based on ASH's estimates of the utilization. The **output** command must be executed before the **stats** command is entered in order to ensure that current statistics are being used.

7.3.3.14 The Subgraph Command

This is the usual ADAS command for moving down the software (or hardware) graph hierarchy. The user must specify the parent node of the subgraph which is to become the current graph.

7.3.3.15 The Window Command

This is the usual ADAS command for changing the window displayed in terms of its center location and relative scaling.

7.4 Output

7.4.1 Mapping Script

ASH will produce an ADAS script file which can be read by GIPSIM to set the appropriate attribute values before simulation. The format for a mapping script is shown in Figure 7.3. The sequence of ADAS commands in the script traverse the graph hierarchy and use the **edit select** command to set the *hw_module* attribute of each leaf software node in the graph hierarchy.

7.4.2 Hardware Module Renaming Script

A typical hardware graph often contains shared subgraphs with the same hardware module names repeated. In order to do the mapping, ASH needs to have a unique name for each hardware module. ASH generates these names by appending a number onto the end of a leaf hardware module name which occurs in a shared subgraph. ASH will produce an ADAS script file which can be read by GIPSIM or ASH to set the appropriate hardware module names before simulation. This renaming must take place if the utilization information generated by ASH is going to be loaded into the hardware graph when running **edigraf**. The format for a hardware module renaming script is shown in Figure 7.4. The sequence of ADAS commands in the script traverse

```
edit select node select hw_module DirOut Memory0 ~
subgraf Sobel
subgraf FV
edit select node select hw_module Z0 TMS0:reg0 ~
edit select node select hw_module Add0 TMS0:calu ~
edit select node select hw_module Add1 TMS0:calu ~
edit select node select hw_module Sub0 TMS0:calu ~
edit select node select hw_module Z1 TMS0:reg1 ~
.
quit nosave
subgraf FH
edit select node select hw_module Z0 TMS0:reg2 ~
edit select node select hw_module Add0 TMS0:calu ~
edit select node select hw_module Add1 TMS0:calu ~
edit select node select hw_module Sub0 TMS0:calu ~
edit select node select hw_module Z1 TMS0:reg3 ~
.
quit nosave
.
quit nosave
edit select node select hw_module Image Memory1 ~
edit select node select hw_module MagOut Memory2 ~
```

Figure 7.3: Example of an ASH Mapping Script Output

the graph hierarchy and use an **edit select node all** command to set the *node_name* attribute of each leaf software node in the graph hierarchy.

7.4.3 Hardware Utilization Script

ASH will produce an ADAS script file which can be read by EDIGRAF or GIPSIM to set the software node utilization attribute values. This allows the user to compare the estimates used by ASH with the actual results generated by a GIPSIM run after the mapping has been completed. The format for a hardware utilization script is shown in Figure 7.5.

7.4.4 Software Utilization Script

ASH will produce an ADAS script file which can be read by EDIGRAF or GIPSIM to set the software node utilization attribute values. This allows the user to compare the estimates used by ASH with the actual results generated by a GIPSIM run after the mapping has been completed. The format for a software utilization script is shown in Figure 7.11.

7.4.5 Log File

ASH will produce a log file which describes the sequence of moves created by ASH. The format for a log is shown in Figure 7.6.

7.5 Example

The example shows the mapping of a flat software graph onto a hierarchical hardware graph with shared subgraphs. The goal for this example is to maximize the throughput of the system, as measured by the utilization of the output nodes. The system will achieve the required throughput rates if the utilization of the output nodes is 100%. This is accomplished by reducing the maximum free utilization of all hardware modules in an effort to minimize $\hat{U}_max$, and thus minimize the difference between $\hat{U}_f(N)$ and $\hat{U}_c(N)$ for all software nodes.

7.5.1 Example Software Graph

The example software graph is a form of the Sobel image processing example. It is shown in Figure 7.7. There are five processes which will be mapped onto memories:

```
edit select node all node_name
PE0
Mem0
PE1
MemBusIn
Mem1
MemBusOut
PEBusIn
PEBusOut
PE2
Mem2
PE3
subgraf PE0
edit select node all node_name
BusIn0
BusOut0
ToBus
CPU0
FromBus
quit nosave
subgraf PE1
. . .
quit nosave
subgraf PE2
. . .
quit nosave
subgraf PE3
. . .
```

Figure 7.4: Example of an ASH Hardware Module Renaming Script Output

```
subgraf PE0
edit status select node select node_utilization BusIn0 M ~
edit select node select node_utilization BusIn0 0.000000 ~
edit status select node select node_utilization BusIn0 P ~
edit status select node select node_utilization BusOut0 M ~
edit select node select node_utilization BusOut0 0.000000 ~
edit status select node select node_utilization BusOut0 P ~
edit status select node select node_utilization CPU0 M ~
edit select node select node_utilization CPU0 0.036068 ~
edit status select node select node_utilization CPU0 P ~
quit nosave
edit status select node select node_utilization Mem0 M ~
edit select node select node_utilization Mem0 0.041617 ~
edit status select node select node_utilization Mem0 P ~
subgraf PE1
. . .
quit nosave
edit status select node select node_utilization Mem1 M ~
edit select node select node_utilization Mem1 0.080460 ~
edit status select node select node_utilization Mem1 P ~
edit status select node select node_utilization PEBusIn M ~
edit select node select node_utilization PEBusIn 0.236623 ~
edit status select node select node_utilization PEBusIn P ~
edit status select node select node_utilization PEBusOut M ~
edit select node select node_utilization PEBusOut 0.359889 ~
edit status select node select node_utilization PEBusOut P ~
subgraf PE2
. . .
quit nosave
edit status select node select node_utilization Mem2 M ~
edit select node select node_utilization Mem2 0.000000 ~
edit status select node select node_utilization Mem2 P ~
subgraf PE3
. . .
quit nosave
```

Figure 7.5: Example of an ASH Hardware Utilization Script Output

```
---Random Mapping---
sfw ImageIn mapped to hdw Mem2 ~
sfw DFilter mapped to hdw CPU1 ~
sfw DirOut mapped to hdw CPU3 ~
sfw CompEDir mapped to hdw CPU0 ~
sfw RowBuf2 mapped to hdw Mem1 ~
sfw VFilter mapped to hdw CPU1 ~
sfw MagOut mapped to hdw Mem0 ~
sfw CompEMag mapped to hdw CPU1 ~
sfw HFilter mapped to hdw CPU2 ~
10---Connectivity---
enforce: CompEDir to CPU1
enforce: DFilter to CPU0
enforce: XRow2 to PEBusIn
enforce: DFilter to CPU3
. . .
sfw ImageIn mapped to hdw Mem1 ~
sfw DFilter mapped to hdw CPU1 ~
sfw DirOut mapped to hdw CPU1 ~
sfw CompEDir mapped to hdw CPU1 ~
sfw RowBuf2 mapped to hdw Mem1 ~
sfw VFilter mapped to hdw CPU1 ~
sfw MagOut mapped to hdw Mem0 ~
sfw CompEMag mapped to hdw CPU1 ~
sfw HFilter mapped to hdw CPU0 ~
sfw RowBuf1 mapped to hdw MemBusIn ~
---Optimization---
optimize: CRow1 to BusOut2
optimize: XRow1 to BusOut3
optimize: HFilter to CPU3
sfw ImageIn mapped to hdw Mem0 ~
sfw DFilter mapped to hdw CPU3 ~
sfw DirOut mapped to hdw CPU0 ~
sfw CompEDir mapped to hdw CPU2 ~
sfw RowBuf2 mapped to hdw Mem1 ~
sfw VFilter mapped to hdw CPU3 ~
sfw MagOut mapped to hdw Mem1 ~
sfw CompEMag mapped to hdw CPU3 ~
sfw HFilter mapped to hdw CPU3 ~
sfw RowBuf1 mapped to hdw PEBusOut ~
```

Figure 7.6: Example of an ASH Log File Output

the input image data store, the magnitude output data store, the direction output data store, and the two scan line buffers (also called row buffers). There are five processes which will be mapped onto processors: the three filters (horizontal, vertical, and diagonal), and the two post processing functions (calculate magnitude and calculate direction). The rest of the processes represent data transfers and are mapped onto bus class hardware modules.

7.5.2 Example Hardware Graph

The hardware graph is hierarchical with two levels. The top level shows a configuration with two buses (one for memory-processor communication and one for inter-processor communication), three memories, and four processing elements. The second level shows the structure of the processing elements, each of which have an internal bus for intra-processor communication.

7.5.3 Example Outputs

The results of an ASH run are three script files:

- A final script which defines the mappings required for the software nodes. This script is shown in Figure 7.10. This script file can be run in GIPSIM to modify the ADAS software graph so that GIPSIM simulations can use the results of ASH.
- Estimated software utilization script. This script can be run against the software file to set the software graph utilization attributes equal to the utilization values estimated by ASH for the final mapping. These utilizations can then be compared with results from a GIPSIM run to understand how close to optimal the ASH mapping results are. This script is shown in Figure 7.11.
- Estimated hardware utilization script. This script can be run against the hardware file to set the hardware graph utilization attributes equal to the utilization values estimated by ASH for the final mapping. These utilizations can then be compared with results from a GIPSIM run to understand how close to optimal the ASH mapping results are. This script is shown in Figure 7.5.

A fourth file that is generated is a log of mapping changes made during each major function of each iteration. This file is shown in Figure 7.6.

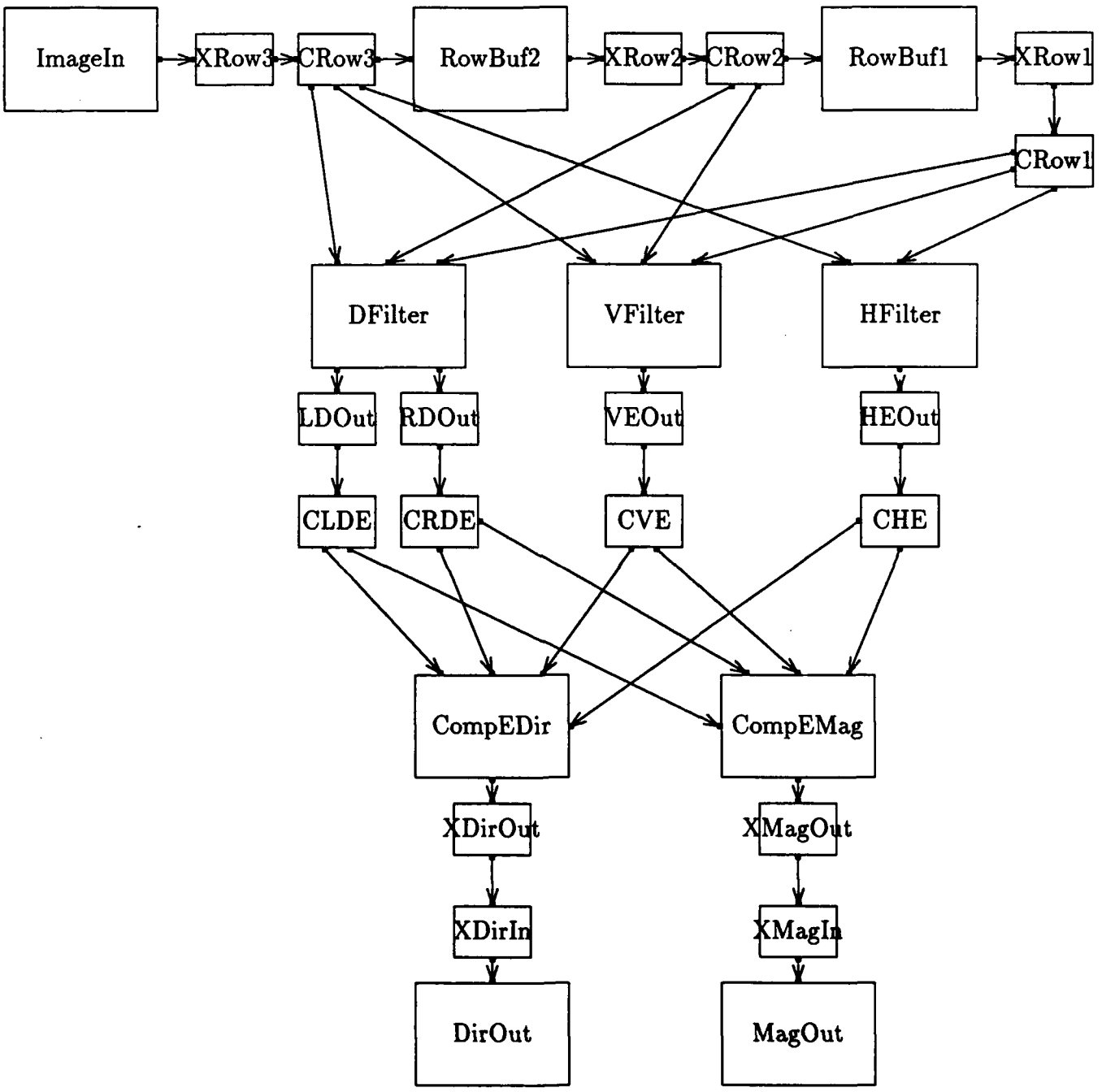

Figure 7.7: Software Graph for the ASH Example

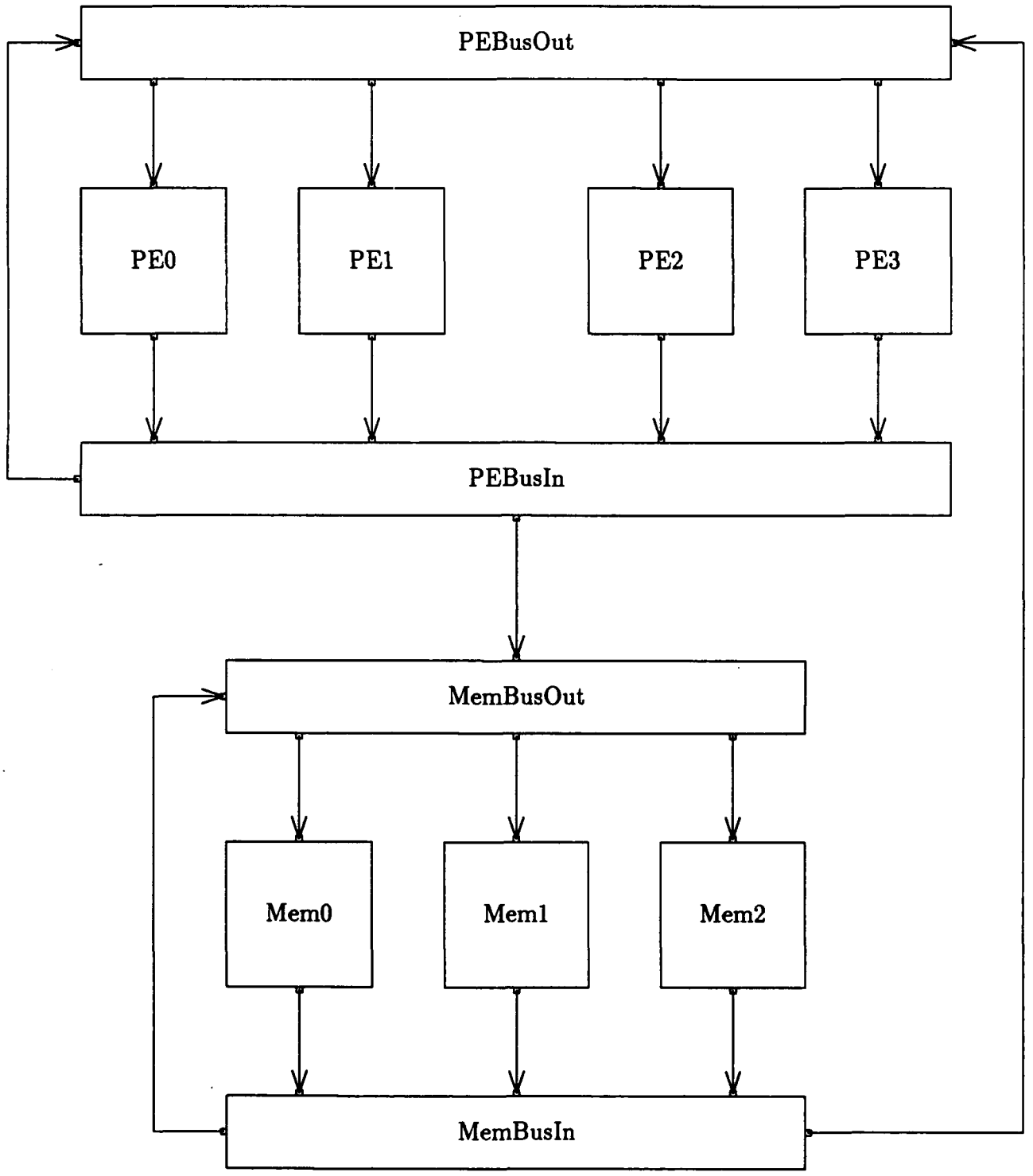

Figure 7.8: Top Level Hardware Graph for the ASH Example

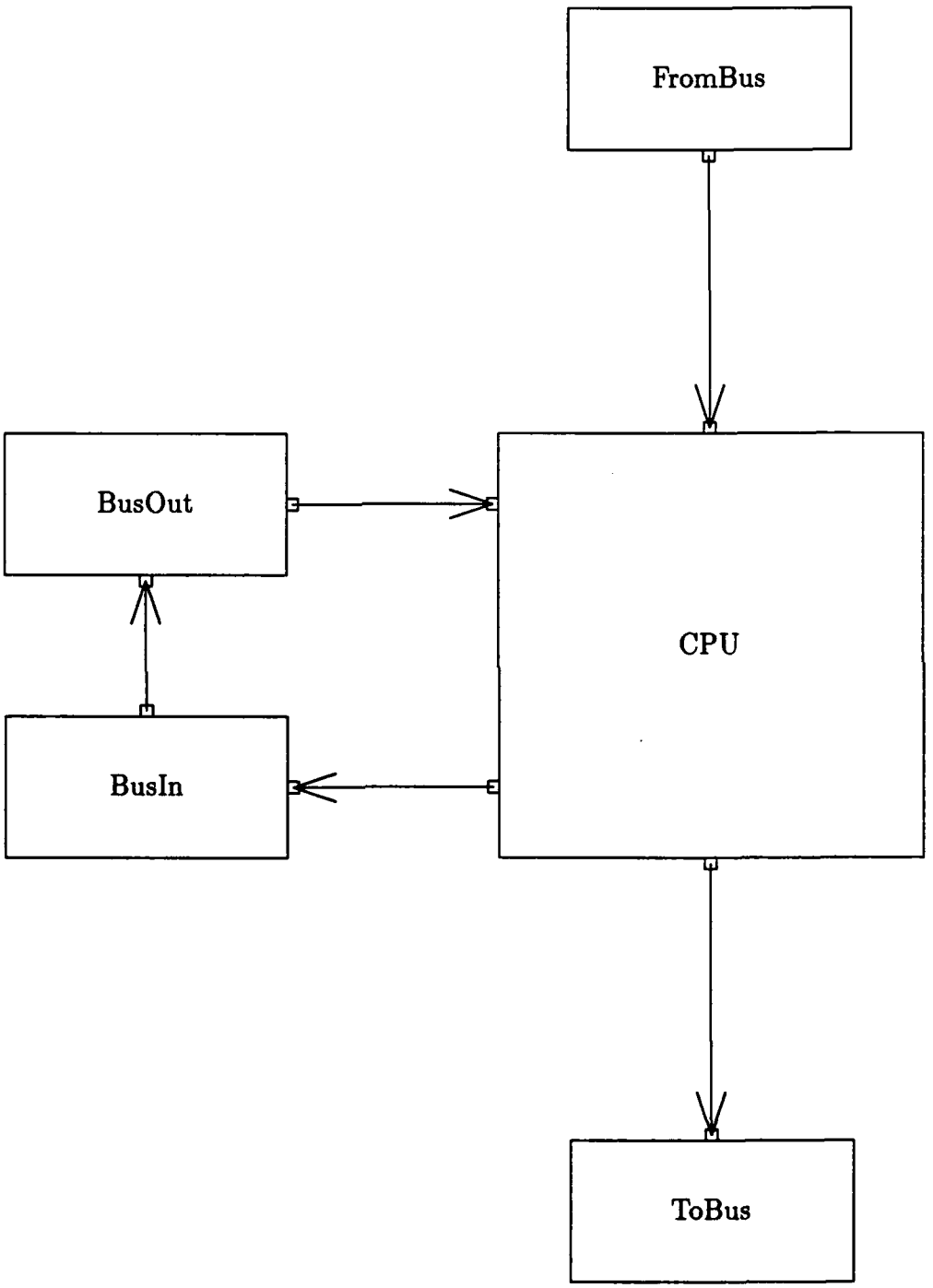

Figure 7.9: Hardware Subgraph for the ASH Example

```
edit select node select hw_module ImageIn Mem0 ~
edit select node select hw_module CRow3P PEBusOut ~
edit select node select hw_module XRow3 MemBusIn ~
edit select node select hw_module CLDE PEBusOut ~
edit select node select hw_module LDOut PEBusIn ~
edit select node select hw_module CRow3M MemBusOut ~
edit select node select hw_module DFilter CPU1 ~
edit select node select hw_module CRDE PEBusOut ~
edit select node select hw_module RDOut PEBusIn ~
edit select node select hw_module DirOut CPU0 ~
edit select node select hw_module XDirIn PEBusOut ~
edit select node select hw_module XDirOut PEBusIn ~
edit select node select hw_module CompEDir CPU3 ~
edit select node select hw_module RowBuf2 Mem1 ~
edit select node select hw_module CVE PEBusOut ~
edit select node select hw_module VEOut PEBusIn ~
edit select node select hw_module VFilter CPU1 ~
edit select node select hw_module CRow2P PEBusOut ~
edit select node select hw_module XRow2 MemBusIn ~
edit select node select hw_module CRow2M MemBusIn ~
edit select node select hw_module MagOut Mem1 ~
edit select node select hw_module XMagIn MemBusOut ~
edit select node select hw_module XMagOut PEBusIn ~
edit select node select hw_module CompEMag CPU1 ~
edit select node select hw_module CHE PEBusOut ~
edit select node select hw_module HEOut PEBusIn ~
edit select node select hw_module HFilter CPU3 ~
edit select node select hw_module RowBuf1 MemBusIn ~
edit select node select hw_module CRow1 PEBusOut ~
edit select node select hw_module XRow1 MemBusIn ~
```

Figure 7.10: Example Mapping Script Output from ASH

```
edit status select node select node_utilization ImageIn M ~
edit select node select node_utilization ImageIn 0.041617 ~
edit status select node select node_utilization ImageIn P ~
edit status select node select node_utilization DFilter M ~
edit select node select node_utilization DFilter 0.440349 ~
edit status select node select node_utilization DFilter P ~
edit status select node select node_utilization DirOut M ~
edit select node select node_utilization DirOut 0.036068 ~
edit status select node select node_utilization DirOut P ~
edit status select node select node_utilization CompEDir M ~
edit select node select node_utilization CompEDir 0.432818 ~
edit status select node select node_utilization CompEDir P ~
edit status select node select node_utilization RowBuf2 M ~
edit select node select node_utilization RowBuf2 0.040824 ~
edit status select node select node_utilization RowBuf2 P ~
edit status select node select node_utilization VFilter M ~
edit select node select node_utilization VFilter 0.242568 ~
edit status select node select node_utilization VFilter P ~
edit status select node select node_utilization MagOut M ~
edit select node select node_utilization MagOut 0.039635 ~
edit status select node select node_utilization MagOut P ~
edit status select node select node_utilization CompEMag M ~
edit select node select node_utilization CompEMag 0.317083 ~
edit status select node select node_utilization CompEMag P ~
edit status select node select node_utilization HFilter M ~
edit select node select node_utilization HFilter 0.202140 ~
edit status select node select node_utilization HFilter P ~
edit status select node select node_utilization RowBuf1 M ~
edit select node select node_utilization RowBuf1 0.040824 ~
edit status select node select node_utilization RowBuf1 P ~
```

Figure 7.11: Example Software Utilization Script Output from ASH

8. ADAS ATTRIBUTE SPECIFICATIONS

Table 8.1 documents AIVD's use of a subset of the ADAS attributes. The first column, *Purpose*, lists the AIVD meaning of the attributes; the second column, *Scope*, lists the ADAS entities with which the attributes are associated; the third column, *Implementation*, lists the current data base attributes that will be used for the prototype. Because *node_user_text* will be used for the node help files, OR-nodes will not be used with examples for this prototype.

Table 8.1: AIVD with the ADAS Attribute Set

Purpose	*Scope*	*Prototype*
Help File	Graph	`graph_user_text`
Help File	Node	`node_user_text`
Help File	Arc	`arc_user_text`
ADL File	Graph	`graph_user_file_name`
ADL File	Node	`node_user_file_name`
ADL File	Arc	`arc_user_file_name`
Node Matching	Node	`graph_port_number`
Inport Matching	Inport	`inport_id`
Outport Matching	Outport	`outport_id`
Arc Matching	Arc	`token_units`